

国际信息工程先进技术译丛

WILEY

用Verilog设计FPGA样机实例解析

FPGA PROTOTYPING BY VERILOG EXAMPLES (XILINX SPARTAN^{TM} -3 VERSION)

[美]　彭皮·楚（PING D.CHU）　著

李艳志　孟伟　刘军　等译





机械工业出版社
CHINA MACHINE PRESS

国际信息工程先进技术译丛

用 Verilog 设计 FPGA 样机实例解析（Xilinx Spartan-3 版）

[美] 彭皮·楚（PONG P. CHU）著
李艳志　孟　伟　刘　军　等译



机械工业出版社

图书在版编目（CIP）数据

用 Verilog 设计 FPGA 样机实例解析：Xilinx Spartan-3 版/（美）楚（Chu，P. D.）著；李艳志等译. —北京：机械工业出版社，2016. 4

（国际信息工程先进技术译丛）

书名原文：FPGA Prototyping By Verilog Examples：Xilinx Spartan-3 Version

ISBN 978-7-111-53644-4

Ⅰ. ①用… Ⅱ. ①楚…②李… Ⅲ. ①可编程序逻辑器件-系统设计 Ⅳ. ①TP332. 1

中国版本图书馆 CIP 数据核字（2016）第 088438 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
策划编辑：林春泉 责任编辑：翟天睿
责任印制：常天培 责任校对：刘秀丽 程俊巧
北京京丰印刷厂印刷
2016 年 11 月第 1 版 · 第 1 次印刷
169mm × 239mm · 36. 25 印张 · 690 千字
0 001—2 000 册
标准书号：ISBN 978-7-111-53644-4
定价：165. 00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务	网络服务
服务咨询热线：010-88361066	机 工 官 网：www. cmpbook. com
读者购书热线：010-68326294	机 工 官 博：weibo. com/cmp1952
010-88379203	金 书 网：www. golden-book. com
封面无防伪标均为盗版	教育服务网：www. cmpedu. com

本书主要内容包括三部分：基本的数字电路、外围模块和内嵌的微控制器。第一部分介绍了基础的HDL结构和对应硬件，并示范如何用这些结构来搭建基本的数字电路。第二部分是应用第一部分的技术为原型板设计外围模块，介绍了一个单独外设的开发、实现和验证。可以将这些模块组成一个复杂的系统。第三部分介绍了基于FPGA的软核微控制器，即PicoBlaze，展示了如何将通用处理器和定制电路进行集成。本书通过实例深入浅出地介绍了使用Verilog对可编程逻辑器件进行设计的方法，不仅介绍了HDL的语法，还重点介绍了对可编程逻辑器件的设计方法，提供了一系列使用Verilog对可编程逻辑器件进行设计的实例，书中的实例均可运行于Xilinx公司的Spartan-3原型开发板中，使读者能够边动手边学习，达到快速入门并掌握其要领的目的。

本书可作为可编程逻辑器件的学习指导书，通过书中的案例，初学者最终可以完全掌握可编程逻辑器件的设计。同时，也可作为工程实践的指导用书，对提高可编程逻辑器件开发人员的设计水平有借鉴价值。

译 者 序

由于可编程逻辑器件具有集成度高、体积小、功耗低、速度快等诸多优点，在航空、航天等军用产品领域中获得了广泛的应用。然而，可编程逻辑器件的开发与以往软件或硬件的开发有着很大的不同，随着大量可编程逻辑器件设计的增加，如何提高开发人员的设计水平成为保障可编程逻辑器件安全性的关键因素。

本书是介绍使用 Verilog 对可编程逻辑器件进行设计的技术著作，主要聚焦在对可编程逻辑器件的设计，而不仅仅是介绍 HDL 的语法。本书提供了一系列使用 Verilog 对可编程逻辑器件进行设计的实例，这些例子可作为通用的基本模块组合成结构复杂的大系统。本书主要内容包括三部分：基本的数字电路、外围模块和内嵌的微控制器。作者 Pong P. Chu 在使用 Verilog 对可编程逻辑器件进行设计方面有多年的工程实践经验。本书以实例为依托深入浅出地介绍了使用 Verilog 对可编程逻辑器件进行设计的方法，并给出很多 Verilog 语言的使用方法，对工程实践有一定的借鉴价值，对提高可编程逻辑器件开发人员的设计水平有一定帮助。

当夏宇闻教授将本书的英文版《FPGA PROTOTYPING BY VERILOG EXAMPLES》推荐给我们时，我们欣喜地发现本书对提高可编程逻辑器件开发人员的设计水平有很大的帮助。当夏宇闻教授建议我们结合工程应用中的实践经验，将本书翻译成中文时，我们立即欣然答应。我们所在的单位——航天科工集团三院三〇四所，主要负责软件和可编程逻辑器件软件的测试及工程化工作，自成立以来承担了院内、空间领域、军用领域上百个型号的测试工作，其中可编程逻辑器件软件的测试工作占 30% 以上，积累了丰富的可编程逻辑设计与验证方面的经验。我们相信中文版的出版将给使用 Verilog 对可编程逻辑器件进行设计的开发人员提供更多的帮助。

参与本书翻译工作的人员有孟伟、李艳志、王俊、于林宇、张津荣、刘军、刘伟、杨楠、王栋、李丽华、赵静、宋悦、荣高峰、田彪、彭鸣、张国宇、高媛、李思、杨豹、黄勇、王春云、朱琳、康文兴、张志刚、周晴、张哲、于润泽、舒毅、陈朋、王颖、张维海、吕宗辉，他们都是从事可编程逻辑器件软件的开发和测试工作的技术人员，具有多年可编程逻辑器件软件的开发和测试工作经验，为探月工程、载人航天工程、空间科学先导专项等国家重大项目中多个型号的顺利完成保驾护航。全书由夏宇闻教授负责审校。

在《FPGA PROTOTYPING BY VERILOG EXAMPLES》中文版即将出版之时，

特别向航天科工集团三院三〇四所对我们翻译工作的支持，向北京航空航天大学夏宇闻教授和机械工业出版社相关编辑的悉心指导和帮助表示衷心的感谢！

鉴于时间紧迫和译者水平有限，书中难免有不当之处，敬请读者批评指正。

译　者　于北京

2016 年 9 月

原 书 序

HDL（硬件描述语言）和FPGA（现场可编程门阵列）器件可以使设计者很快地完成复杂数字电路的开发和仿真，并在样机器件上实现，随后对器件电路的实际运行情况进行检查。随着工艺的成熟，HDL和FPGA已经成为设计实践的主流。利用PC和普通的FPGA开发板就能构造出十分复杂的数字系统。本书采用实际操作的学习方法，利用丰富的示例来阐述FPGA和HDL的开发和设计过程。书中包含了大量实例，从简单的门级电路，到带有8位软核处理器和定制I/O外设的复杂嵌入式系统。所有这些例子都可以被综合成具体电路，并在开发板上进行实际测试。

本书关注重点

本书关注的重点是综合后生成硬件的优劣，而不是HDL语法。本书只关注一小部分可综合子集，并使用少量的代码模板为不同类型的电路提供框架，而不是解释每一个语句的结构。这些模板都是通用的，很容易综合到复杂的系统中。虽然这种方法限制了语法表达的“自由”，但并不妨碍我们开发创新性的硬件结构。由于HDL语言的通用性和适应性，同一个电路通常可以用多种语言结构和代码风格表达。其中许多代码是用于建模的。这些代码综合后可能导致不必要的复杂硬件实现，有时还根本不可能综合成任何具体电路。这种模板方式实际上能够促使我们更多地去思考硬件电路本身，养成良好的编码习惯。由于我们的主要兴趣是在硬件上，所以花一些时间研究如何使用同一个代码模板来开发多种不同的硬件结构，而不是用多种不同版本的代码来描述同一个电路是十分有价值的。

目前有两种流行的HDL语言，它们分别是VHDL和Verilog。这两种语言都得到广泛的应用，并且都是IEEE标准。本书使用Verilog，而另一本标题类似的书使用VHDL。尽管两者的语法差异较大，但它们的功能却非常相似，都能很好地达到设计目标。当我们掌握了一种语言的设计实践和编码方法后，再学习另一种语言就会变得非常简单。

虽然本书是为初学者编写的，但书中的示例都严格遵循设计准则，可为读者今后的工作打下良好的基础。编码和设计方法是“向上兼容的”，意思是：

- 同样的方法可应用于未来的大型设计之中。
- 同样的方法能够有助于其他的系统开发任务，包括仿真、时序分析、验证和测试。
- 同样的方法能够被应用到ASIC技术和不同类型的FPGA器件中。

● 代码能够被不同厂家的综合软件综合。

总之，本书是一本实用的、以硬件为核心的教材，其内容涉及用最简洁的 HDL、遵循规范的设计和编码原则，最大限度地实现向上兼容。

购买本书的益处

本书包含三大部分：

基本数字电路、外设模块和嵌入式微控制器。针对的读者群除了正在学习入门级或者高级数字系统设计课程的学生外，还包括想要学习 FPGA 和基于 HDL 开发的在职工程师。对于书中前两部分内容，需要读者具有数字系统的基本知识，而数字系统通常是电子工程和计算机工程专业课程中的必修课。对于第三部分的内容而言，如果读者之前学习过汇编语言编程将会很有帮助。

本书所采用的工具

虽然本书的主要目标是教会读者如何编写与开发工具和 FPGA 器件无关的 HDL 代码，但我们必须得选择一种开发工具（即软件包）和一套 FPGA 开发板来进行综合和实现这些示例。本书使用了 Xilinx 公司的综合工具和 FPGA 器件，Xilinx 公司是一家在该领域处于领导地位的公司。

软件

使用的综合软件是 Xilinx ISE 开发套件的网络版。与完全版的套件相比，网络版除了支持的器件数量受限外，功能与完全版是类似的。大多数入门级开发板都使用便宜的 Spartan-3 系列 FPGA 器件。由于网络版支持 Spartan-3 器件，因此它符合我们的需求。本书使用的仿真软件是 Mentor Graphics 公司的 ModelSim XE Ⅲ入门版。它是 ModelSim 的定制版。这两个软件包都是免费的，并且能够从 Xilinx 网站上下载。

FPGA 开发板

使用由 Digilent Inc 公司生产的几款入门级 FPGA 开发板，其中包括 Spartan-3 Starter、Nexys-2 和 Basys 等开发板，它们都包含一个 Spartan-3/3E FPGA 芯片和相似的外围电路。书中的设计示例是基于 Spartan-3 Starter 开发板（或简称为 S3 板）的，但大多数示例也能在其他开发板上直接使用。HDL 代码的适用性总结如下：

● Spartan 3 Starter（S3）开发板。S3 开发板包含所有的外围器件，不需要其他的附属模块。所有的 HDL 代码和相关讨论能够直接应用在这块开发板上。

● Nexys-2 开发板。Nexys-2 开发板是一块比较新的开发板，包含了一片较大型的 FPGA 芯片和存储芯片。外围器件与 S3 开发板相似。与 S3 开发板有两处不同：第一，其 VGA 接口的“颜色深度”由 3 位扩展到 8 位。因此，第 13 章和第 14 章讨论的 VGA 接口电路的输出需要根据情况进行相应的修改。第二，Nexys-2 板包含更复杂的外部存储器。尽管能够配置为异步 SRAM，但其时序特性与

S3 开发板上的存储芯片是不同的，因此第 11 章描述的存储控制器 HDL 代码便不能直接使用。但同样的设计准则依然可以应用于新的控制器的构造。

- Basys 开发板。Basys 开发板是一个简易的开发板。缺少 RS-232 连接器。为了实现第 8 章描述的 UART 模块和串行接口，我们需要 Digilent 的 RS-232 转换器外设模块。Basys 开发板没有外部存储器件，因此第 11 章描述的存储控制器无法应用。

- 其他 FPGA 开发板。本书描述的大部分外设器件其实都是工业级标准的器件，并且相应的 HDL 代码能够被用于开发板上，只要开发板提供类似的接口和连接器。除了 Xilinx 特定的部分外，这些代码也能够应用于其他厂商设计的基于 FPGA 器件的开发板上。

PC 的附件

设计示例包含了用于连接 PC 外围设备的接口。键盘、鼠标和 VGA 显示器是必需的，还需要一条用于连接 UART 模块的普通串行数据线。这些外设使用广泛，通常可以在一台旧的 PC 上找到。

本书的结构

本书分为 3 个主要部分。第一部分介绍了基础的 HDL 结构和对应硬件，并示范如何用这些结构来搭建基本的数字电路。本部分由 7 个章节组成：

- 第 1 章介绍了 HDL 程序的结构、基础语法和逻辑操作符。根据这些语言结构，可以推导出相应的门级组合电路。

- 第 2 章介绍了 FPGA 器件、原型板和开发流程。借助于 Xilinx ISE 综合软件教程和 Mentor Graphics ModelSim 仿真软件教程进行开发过程的示范。

- 第 3 章介绍了与 HDL 语言相关的操作符和算法操作符及其电路的结构。它们与中规模元件（如比较器、加法器和多路复用开关等元件）对应。模块级组合电路就是由这些语言结构得到的。

- 第 4 章介绍了存储元件和构造简单时序电路的代码，例如计数器和移位寄存器，这些电路的状态转移表现为简单的有序模式。

- 第 5 章讨论了有限状态机（FSM）的构建，有限状态机也是一种时序电路，但它的状态转移表现为复杂的非有序模式。

- 第 6 章介绍了带有数据路径的有限状态机（FSMD）的构建。FSMD 用于实现寄存器传输（RT）方法学，通过数据在寄存期间的传输和操作，描述系统的运行。

- 第 7 章介绍了关于语言构造和编码技术方面的若干高级话题，并介绍了更加复杂测试平台的开发技术。读者可以跳过该章，不会影响对其他章节的理解。

第二部分是应用第一部分的技术为原型板设计一系列外围模块。每章介绍一

个单独外设的开发、实现和验证。可以将这些模块组成一个更大的工程。该部分包括 7 个章节：

● 第 8 章介绍了普通的异步收发送器（UART）的设计，用于提供原型板上的 RS-232 接口接收和发送数据的串行链路。

● 第 9 章介绍了键盘接口的设计，可以从键盘上读取扫描码。键盘通过开发板上的 PS2 接口与其连接。

● 第 10 章介绍了鼠标接口的设计，可以从鼠标上获得点击和移动信息。鼠标也是通过开发板上的 PS2 接口与其连接。

● 第 11 章讨论了存储控制器的实现和时序。该控制器用于对 S3 板上的两个静态随机存储器（SRAM）读取和写入数据。

● 第 12 章讨论了 Spartan-3 器件中特定元件的推断和使用。重点是 FPGA 的内部存储块。

● 第 13 章介绍了一个视频控制器的设计和实现。讨论的内容包括视频同步信号的产生并展示了比特映射和对象映射图像界面的构造。显示器通过开发板上的 VGA 接口进行连接。

● 第 14 章继续介绍视频控制器的开发。讨论展示了文字界面和常规分片映射机制的构建。

第三部分介绍了基于 FPGA 的软核微控制器，即 PicoBlaze，展示了如何将通用处理器和定制电路进行集成。该部分包括 4 个章节：

● 第 15 章对 PicoBlaze 的结构和指令集进行了简介。

● 第 16 章对基本的汇编语言编程进行了介绍，并提供了一个总的开发流程。

● 第 17 章讨论了 PicoBlaze 的 I/O 特性，并展示了如何将其与其他外设通过定制电路连接起来。

● 第 18 章讨论了 PicoBlaze 的中断性能，并展示了一个定制的中断处理电路的构建。

除了常规的章节，附录部分总结和列出了所有的代码模版。

特殊标识[xilinx specific]　本书中我们使用了两种特殊的段落标记：一个是为了描述 Xilinx-specific（Xilinx 公司特有的）特性，另一个是为了描述 Verilog-1995 的结构。虽然在本书中描述的例子是基于 Xilinx 开发板来实现的，并且代码也是采用 Xilinx ISE 软件进行综合，我们仍设法使 HDL 代码不依赖于器件和软件。本书提到的大多数内容和代码能够应用到不同的目标器件，也可以被不同的综合软件综合。然而，一些代码或器件特性是 Xilinx ISE 或者 Spartan-3FPGA 芯片所独有的。我们用 Xilinx specific 上标，表示相应的部分或者章节的内容仅是针对 Xilinx 公司的器件。

同样，像在这一页的边缘，我们用边缘标记来表明这段的内容仅针对 Xilinx。这些标记表明代码或设计不可直接移植，需要针对不同的软件包或目标板对代码和设计进行修改。1995 年 Verilog 语言第一次得到批准（被引用作 Verilog-1995），并在 2001 年修订（被引用作 Verilog-2001）。修订版有很多改进。本书中使用的是 Verilog-2001。如果一种语言使用了两种不同版本进行构建，我们会将其分开，单独对旧版本进行描述，并在页边缘做上标记，用于这种类型的讨论。这些内容“供参考”，目的是帮助读者理解旧版本的 Verilog 代码。

指导价值

本书可作为数字系统概论或者高级工程指导的参考书。在数字系统概论中，本书提供了课程的试验部分。第一部分的章节基本上都遵循了典型的课程顺序，可作为常规课程的参考。可以选择一两个外设模块作为案例进行学习，相应的实验可作为学期实验。

在高级工程指导课程中，本书提供了独立开展工程的基础。在第一部分的章节可以看作是复习资料，里面提供了 HDL、综合和 FPGA 开发板的基本背景。第二部分的一些模块示例可用于示范更复杂的电路设计。这些模块也可以看作是基本模块（即 IPs）或集成到最终工程的子系统。如果需要设计一个嵌入式系统，那么第三部分讨论的 PicoBlaze 微控制器可用作一个通用处理器。

相关网站

本书的配套网站（http://acadernic. csuohio. edu/chu_p/rtpl）提供了其他信息，包括以下内容：

- 勘误表；
- 编码模型；
- HDL 代码示例和相关文档；
- 仿真和综合软件的下载链接；
- 参考资料的下载链接；
- 其他工程概念。

勘误表　本书为自编书籍，即作者完成了本书的所有部分，包括示例、图表、代码示例、索引和格式。错误的出现不可避免，因此本书的配套网站提供了最新的勘误表，并提供了错误报告。

P. P. Chu

克利夫兰，俄亥俄州

2008 年 1 月

致　谢

感谢 George L. Kramerich 教授的鼓励和帮助。同时感谢 John Wiley & Sons 等。感谢他们允许本书使用《RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability》一书3.1、3.2、4.2、4.10、4.11、6.5和7.2节的插图，感谢 Xilinx 公司允许本书使用 Spartan-3 Starter 开发板使用说明中的2.3和9.3插图。所有在本书中使用或引用的商标的所有权均为它们各自拥有者所有。

P. P. Chu
克利夫兰，俄亥俄州
2008年1月

目　录

第一部分　基本数字电路

第二部分 I/O 模块

第一部分

基本数字电路

第 1 章　门级组合电路

1.1　简介

Verilog 语言是一种硬件描述语言，产生于 20 世纪 80 年代中期，后成为了 IEEE（美国电气电子工程师学会）标准，即 IEEE-1364 标准。该标准于 1995 年得到批准（即 Verilog-1995），并于 2001 年进行了修订（即 Verilog-2001）。修订版有很多改进。本书中使用的是 Verilog-2001。

Verilog 用于在不同级别上对数字系统描述和建模，是一种极其复杂的语言。本书重点在于硬件设计而并非语言本身。我们将通过分析和研究一系列实例来介绍关键的 Verilog 综合结构，而不是覆盖 Verilog 语言的每一方面。一些高级主题将在第 7 章加以分析，详细的 Verilog 全方位使用可通过本章最后列出的参考文献部分加以探究。

尽管 Verilog 的语法在某种程度上与 C 语言类似，它的语义（即“意义”）却是以并发的硬件操作为基础，这与 C 语言的顺序执行完全不同。一些语言构造上的细微的差别和一些 Verilog 语言所固有的非确定性行为能够导致难以检测的错误并引入仿真和综合之间的差异。本书的代码遵循“安全比有问题更重要”原则，不使用速写和短代码，而重点考虑代码风格及构造结构是否清晰可综合并能精确描述想要的硬件。

在本章，我们用一个简单的比较器来说明一段 Verilog 程序的框架。该描述只使用逻辑运算符描述了一个由简单的逻辑门构成的门级比较电路。在第 3 章，我们再介绍余下的 Verilog 运算符及结构，并分析由加法器、比较器、乘法器等中等规模单元组成的 RTL（寄存器传输级）组合电路。

1.2　一般描述

考虑有两个输入 i0 和 i1 和一个输出 eq 的 1 位比较器。当 i0 和 i1 相等时信号 eq 置位。该电路的真值表如表 1-1 所示。

表 1-1　一位比较器真值表

输入		输出结果
i0	i1	
0	0	1
1	0	0
1	0	0
1	1	1

假设我们想要使用非（not）、与（and）、或（or）及异或（xor）运算的基本逻辑门来实现该电

路，一种描述电路的方法就是使用乘积项的形式。逻辑表达式为

$$eq = i0 \cdot i1 + i0' \cdot i1'$$

一种可行的 Verilog 代码如示例 1.1 所示。我们在下面的小节中分析语言结构和代码。

示例 1.1　一位比较器的门级实现

```
module eq1
   //I/O 端口
   (
   input wire i0,i1,
   output wire eq
   );
   // 信号声明
   wire p0,p1;
   // 程序体
   // 两个乘积和
   assign eq = p0 | p1;
   // 乘积
   assign p0 = ~i0 & ~i1;
   assign p1 = i0 & i1;
endmodule
```

理解一段 HDL(硬件描述语言)程序最好的办法是将其想象成一个硬件电路。上述程序包含 3 个部分。I/O 端口部分用于描述该电路的输入/输出端口,分别为 i0、i1 及 eq。信号声明部分指定了内部连接信号,分别为 p0 和 p1。程序体部分描述了电路的内部组织结构。此段代码中有 3 个连续赋值语句。每一个都可以作为执行特定的简单逻辑操作的电路。在接下来的章节我们将会详细分析语言结构和代码。

该段程序的图形表示如图 1-1 所示。3 个连续赋值组成 3 个电路。它们之间的连接是通过信号和端口名隐式指定的方法完成的。

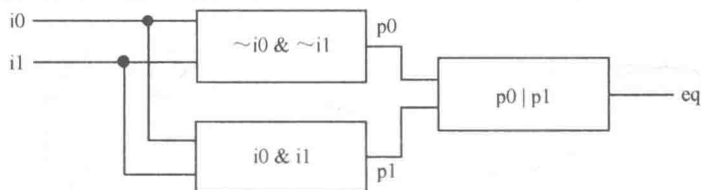


图 1-1　比较器程序的图示

1.3 基本词汇元素

标识符 标识符赋予一个对象唯一的名字，如 eq1、i0 或者 p0。由字母、数字、下划线(_)及符号（$）组合。$通常与系统任务或函数一起使用。

标识符的首个字符必须是字母或下划线。一个很好的习惯是使用描述性的名字为对象命名。例如，对于一个存储器地址使能信号来说，mem_addr_en 要比 mae 更加容易理解。

Verilog 是一种字母大小写敏感的语言。因此，data-bus、Data-bus 和 DATA_BUS 是 3 个不同的对象。为避免混淆，我们应尽量不要通过使用字母大小写的方式创造不同的标识符。

关键字 关键字是指用于描述语言结构的预定义标识符。本书中的 Verilog 关键字使用黑体字，比如示例 1.1 中的 **module** 及 **wire**。

间隔符 间隔符在 Verilog 代码中用于分隔标识符，包含空格符、制表符及换行符。使用恰当的空格符来安排代码格式使其更具有可读性。

注释 注释仅用于记录，因而会被软件忽略。Verilog 有两种格式的注释。单行注释以//开始，例如

```
//This is a comment
```

多行注释封装在/＊和＊/之间，例如

```
/* This is comment line 1.
   This is comment line 2.
   This is comment line 3. */
```

在本书中，我们使用斜体字来书写注释，如上述的例子。

1.4 数据类型

1.4.1 四值系统

四种基本数值适用于 Verilog 大部分数据类型：

- 0：表示逻辑 0，或条件为假；
- 1：表示逻辑 1，或条件为真；
- Z：表示高阻态；
- X：表示不确定值。

Z 值相当于三态缓冲器的输出。X 值通常用于建模和仿真，表示一种非 0、1 或 Z 的值，例如一种未初始化的输入或输出冲突。

1.4.2 数据类型分类

Verilog 主要有两种数据类型：线网和变量。

线网类　该数据类型表示硬件元件间的物理连线。用于作为连续赋值的输出及不同模块间的连接信号。最常使用的数据类型是 wire 型。顾名思义，它表示一条连线。

wire 数据类型表示 1bit 信号，如

```
wire  p0,p1;    //两个1bit 信号
```

当把信号的集合划为一条总线时，我们使用一维数组（向量）来表示，如

```
wire  [7:0]  data1, data2; //8bit 数据
wire  [31:0]  addr;        //32bit 地址
wire  [0:7]  revers_data;  //应避免升序索引
```

索引范围既可以降序排列（如[7:0]）也可升序排列（如[0:7]），前者使用更加广泛是因为最左边的位置（也就是第 7 位）与二进制数最高有效位一致。

有时需要使用二维数组表示一个存储器。例如，一个 32 ×4 的存储器（也就是一个存储器有 32 个字而每个字为 4bit 位宽）可以表示为

```
wire  [3:0]  mea1  [31:0];//32 乘4 寄存器
```

线网类中另一些数据类型表示特定的逻辑行为或功能，例如 wand（用于线与连接）及 supply0（用于电路接地连接）。本书不会使用这些数据类型。同时，Verilog-2001 也允许使用 singed 型，该类型将在本书 7.3 节讨论。

变量类　变量类中的数据类型表示行为模型中的抽象存储，用于过程赋值的输出。变量类包含五种数据类型：reg、integer、real、time 以及 realtime。最常用的数据类型是 reg，它是可综合的。推断出来的电路可能有也可能没有物理存储元件。后三种数据类型只在建模及仿真中使用，integer 数据类型在本书 7.3 节讨论。

在 Verilog-1995 中，变量类就是我们所知道的寄存器类型。由于该术语与物理硬件寄存器（也就是触发器的集合）相同，为了避免混淆，Verilog-2001 文档中进行了改变。在本书中，我们使用变量表示数据类型，而使用寄存器表示物理寄存器电路。

1.4.3 数字表示

在 Verilog 中，一个整型常数可以用多种格式表示。通常的表示为

[符号][位宽]'[基数][数值]

[基数]指定数字的进制，可以为以下形式：

- b 或 B：二进制；

- o 或 O：八进制；
- h 或 H：十六进制；
- d 或 D：十进制。

[数值]指定对应基数下的数值。为了表达清晰，数值中可以包含下划线(_)。[位宽]指明了数字的比特位数，它是可选的。当[位宽]存在时，数字是有具体位宽限制的，否则无具体限制。

指定位宽数字　一个指定位宽的数字明确指定了比特宽度。除一些特殊情况外，如果数值本身所需的位宽比[位宽]指明的小，就在前端填充零，用以扩展数值。如果最高有效位是 z 或 x，则填充 z 或 x 值，而当使用有符号的数据类型时，则需要填充最高有效位。一些有位宽数的范例在表 1-2 顶端加以介绍。

未指定位宽数字　未指定位宽数字省略了[位宽]项。实际的位宽取决于主机但至少为 32bit。若数值为十进制，则'[基数]项也可以省略。假设主机使用 32bit 位宽，几个无指定位宽数字的例子如表 1-2 底部所示。

表 1-2　有位宽数和无位宽数

数值	存储值	备注
5'b11010	11010	
5'b11_010	11010	忽略
5'o32	11010	
5'h1a	11010	
5'd26	11010	
5'b0	00000	拓展 o
5'b1	00001	拓展 o
5'bz	zzzzz	拓展 z
5'bz	xxxxx	拓展 x
5'bx01	xxx01	拓展 x
-5'b00001	1111	2's complement of 00001
b'11010	00000000000000000000000000011010	拓展至 32bit
'hee	00000000000000000000000011101110	拓展至 32bit
1	00000000000000000000000000000001	拓展至 32bit
-1	11111111111111111111111111111111	拓展至 32bit

1.4.4　运算符

Verilog 语言有二十多种运算符。对于门级描述，我们只需使用下列按位运算符：~（非）、&（与）、|（或）及^（异或）。这些运算符用于推断基本门

级单元。其他运算符在本书 3. 2 节讨论。

1. 5 程序结构

正如名字所标明的，HDL 用于描述硬件。当我们开发或检查一段 Verilog 代码时，如果我们以“硬件结构”来思考要比“按序算法”更容易领会。本书中的大部分 Verilog 代码依据示例 1. 1 中所示的基本架构。其由三部分构成：I/O 端口声明、信号声明及模块体。

1. 5. 1 端口声明

示例 1. 1 中的模块声明和端口声明为

```
module eq1
(
  input  wire i0, i1,
  output  wire eq
);
```

I/O 声明指明了方式、数据类型及模块 I/O 端口名称。

简化的语法为

```
module [module-name]
(
[mode] [data-type] [port-names],
[mode] [data-type] [port-names],
...
[mode] [data-type] [port-names]
);
```

[mode]项可以为输入型、输出型、或者输入/输出型，分别代表输入、输出，或者双向端口。注意在最后的声明后没有逗号。若数据 wire 型则[data_type]荐可以省略。

Verilog-1995 端口声明 在 Verilog-1995 中，端口名称、类型、数据类型是分别进行声明的。例如，前文所述的端口声明应变为

```
module eq1 (i0, i1, eq); // 括号内只有端口名字
  // 声明方式
  input i0, i1;
  output eq;
  // 声明数据类型
```

```
wire i0, i1;
wire eq;
```

在本书中我们不使用这种形式。

1.5.2 程序体

在C语言中语句是顺序执行的，而与C语言不同的是，一个可综合的Verilog模块可以认为是一些电路的集合。这些部分并行操作并发执行。描述一个部分可以有许多方法：

- 连续赋值；
- “Always块”；
- 模块示例。

描述一个线路部件的第一种方法是使用连续赋值。这对简单的组合电路很有帮助。其简单表示为

```
assign [signal_name] = [expression];
```

每个连续赋值都可以认为是一个电路部件。左手边的信号为输出，而右手边表达式中的信号为输入。表达式描述了电路的功能。例如语句

```
assign eq = p0 | p1
```

这是执行逻辑或操作的电路。当p0或p1的值发生变化，则激活该条语句求值。在传播延迟后eq赋新值。在示例1.1中有三种连续赋值，它们与图1-1中的3个电路相一致。由于赋值与3个电路相协调，因此这些语句的顺序不重要。

描述一个电路部件的第二种方法是使用always块。更多理论的程序分配在always块内使用，因此它能够用来描述更加复杂的电路操作。always块将在本书3.3节讨论。

描述一个电路部件的第三种方法是使用模块例化。例化中创建了另一个模块并允许我们合并预先写好的模块作为当前模块的子系统。在本书1.6节将对例化进行讨论。

1.5.3 信号声明

声明部分指定了模块使用的内部信号和参数。内部信号可以认为是电路部件间的互接连线，如图1-1所示。

简单的信号声明句法为

[数据类型][端口名称]

示例1.1中声明了两个内部信号：

```
wire p0, p1;
```

隐式线网 在Verilog中，标识符不需要明确声明。如果一个标识符是可忽

略的，那么它就被定义为隐式线网。错误的数据类型是 wire 型。我们可以移动示例 1.1 中的外部声明，而简单代码在示例 1.2 中展示。

示例 1.2　带有隐式线网的代码

```
module eq1_implicit
  (
    input i0, i1,              // 无数据类型声明
    output eq
  );
// 无内部信号声明
// 乘积项必须在前面放置
assign p0 = ~i0 & ~i1;         // 隐式线网
assign p1 = i0 & i1;           // 隐式线网
// 两个乘积项的和
assign eq = p0|p1;
endmodule
```

尽管代码变得更加简洁，但是这样做会引入标识符拼写错误所带来的小错误，在本书中，我们通常使用外部声明。

1.5.4　其他例子

我们可以将比较器拓展到 2bit 输出。输入为 a 和 b，而输出为 aeqb。当 a 和 b 的各位都相等时 aeqb 置为有效。代码如示例 1.3 所示。

示例 1.3　2bit 比较器门级执行

```
module eq2-sop
  (
  input wire [1:0] a,b,
  output wire aeqb
  );
// 内部信号声明
  wire p0,p1,p2,p3;
  // 乘积项的和
  assign aeqb = p0|p1|p2|p3;
  // 乘积项
```

```
    assign p0 = ( ~a[1]& ~b[1])&( ~a[01&-~b[0]);
    assign p1 = ( ~a[11& ~b[1])&(a[0]&b[0]);
    assign p2 = (a[1]&b[1])&( ~a[0]& ~b[0]);
    assign p3 = (a[1]&b[1])&(a[0]&b[0]);
endmodule
```

a 和 b 端口声明为两元素数组。结构体的导出与 1bit 比较器的推导相似。信号 p0、p1、p3 及 p3 代表 4 个乘积项的结果，而最终的结果 apeqb 是乘积和形式的表达式。

1.6　结构描述

一个数字系统通常由一些较小的子系统构成。这就允许我们用更简单的或者预定好的元件来构建一个大的系统。Verilog 中提供了模块例化的机制来执行此任务。这种类型的代码叫做结构描述。

可选择的设计一个 1.5.4 节中的 2bit 比较器的方法是利用先前构造的 1bit 比较器作为构造块。图 1-2 中，用两个 1bit 比较器来检测两个单独的 bit 而它们的结果被反馈为一个逻辑与单元。只有当 bit 为均相等时 aeqb 信号有效。相应的代码如示例 1.4 所示。

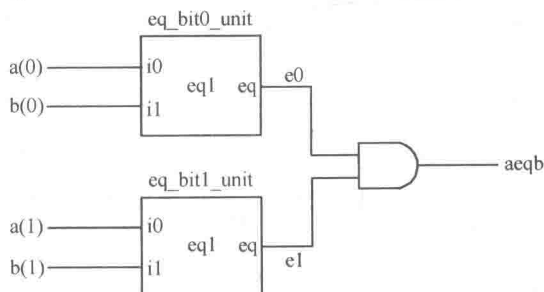


图 1-2　从 1bit 比较器到 2bit 比较器的结构

示例 1.4　2-bit 比较器的结构描述

```
module eq2
  (
    input wire [1:0] a, b,
    output wire aeqb
  );
  // 内部信号声明
```

```
  wire e0, e1;
  // 实体
  // 例化两个1bit 比较器
  eq1 eq_bit0_unit (.i0(a[0]),.i1(b[0]),.eq(e0));
  eq1 eq_bit1_unit (.eq(e1),.i0(a[1]),.i1(b[1]));
  // 若每位都相等则a 与b 相等
  assign aeqb = e0 & e1;
endmodule
```

代码包含两个模块例化语句。简单化的模块例化语法为

```
[模块名称][例化名称]
(
.[端口名称]([信号名称]),
.[端口名称]([信号名称]),
...
);
```

语句的第一部分指定使用哪一个元件。[模块名称]指出模块的名称，而[例化名称]给出了一个例化的独有身份。第二部分是端口连接，显示一个例化模块（低级模块）I/O 端口间的连接及当前模块（高级模块）中使用的外部信号。这种映射的形式是依据名称的连接。而端口名称及信号名称的顺序没有关系。

在示例 1.4 中，第一条元件例化语句为

```
eq1 eq_bit0_unit(.i0(a[0]),.i1(b[0]),.eq(e0));
```

eq1 为示例 1.1 中定义的模块名称。端口映射反映了图 1-2 中所示的连接。元件例化语句代表了一个“黑盒”中的电路，而这个“黑盒”的功能在另一个模块中定义。

这个例子证明了一个块图表与代码间的紧密联系。代码本质上是一段示意图的文本描述。对于人员理解一个图表来讲，这是种比较笨拙的方法，它是将所有的表示都放入一个单独的 HDL 框架中。Xilinx ISE 包中含有一个简单的示意性的编译效用，它能够抓取图解形式中的示意细节，然后将图表转换为一段 HDL 结构描述。

通过规则列表连接　联系端口和外部信号的可选择的方式是通过规则列表（有时也称为通过位置连接）。在这种方式中，低级模块的端口名称可以忽略，而高级模块的信号用和低级模块端口定义时的相同顺序列出。通过这种方式，在示例 1.4 中的两个模块例化语句可以重新写成

```
eq1 eq_bit0_unit (a[0],b[0],e0);
eq1 eq_bit1_unit (a[1],b[1],e1);
```

尽管这种方法使得代码更简洁，但是它可能会产生错误，特别是对于一个有很多I/O端口的模块。例如，如果我们更改低级模块的代码并转换端口声明中两个端口的顺序，那么例化的模块也都需要更正。如果在代码编译期间发生变化，那么改变的端口顺序可能在综合时未被发现遗留，从而导致难以发现的程序缺陷。

Verilog 原语 Veriolg 包含一系列可以作为模块例化的预先设定的原语。这些原语符合简单的门级功能，如 and，or 及 not 单元。例如，eq1 电路可以通过使用简单单元实现，如图 1-3 所示。相应的原语基本代码如示例 1.5 所示。

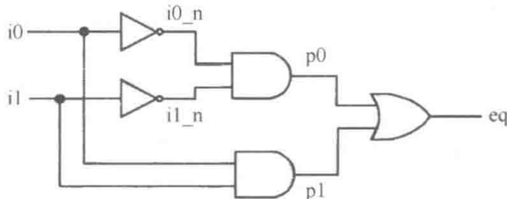


图 1-3 2bit 比较器低级图表

示例 1.5 用 Verilog 原语执行

```
module eq1-primitive
   (
    input wire i0, i1,
    output wire eq
  )
  // 内部信号声明
  wire i0_n, i1_n, p0, p1;
  // 原语门实例化
not unit1 (i0_n, i0); //i0_n = ~i0;
not unit2 (i1_n, i1); //i1_n = ~i1;
and unit3 (p0, i0_n, i1_n); //p0 = i0_n & i1_n;
and unit4 (p1, i0, i1; //p1 = i0 & i1;
or unit5 (eq, p0, p1); //eq = p0 | p1;
  endmodule
```

代码的该种格式非常冗长并且可以用简单的位逻辑运算符所代替。在本书中我们不使用原语。

除预定原语之外，我们也可以定义定制原语，也就是用户数据报协议（UDPS）。

例如，我们可以在一个 UDP 中定义一个 1bit 比较器电路，如示例 1.6 所示。

示例 1.6　1bit 比较器用户数据报协议

```
primitive eq1-udp(eq, i0, i1);
  output eq;
  input i0, i1;

table
// i0 i1 :eq
   0  0  : 1 ;
   0  1  : 0 ;
   1  0  : 0 ;
   1  0  : 1 ;
endtable
endprimitive
```

一个用户数据报协议本质上是一个电路的表描述。同样的表也可以用一条 case 语句描述（3.5 节讨论）。在本书中我们使用后面的方法而不使用用户数据报协议。

1.7　测试平台

一旦代码建立起来，就可以用主机仿真来验证一个电路操作的正确性并且代码会被综合成一个物理设备。模拟是在相同的 HDL 框架中执行。我们创造一种特殊的程序 testbench，来模仿一个物理实验平台。2bit 比较器的 testbench 如图 1-4 所示。在测试装置为测试底层单元，测试向量生成程序产生测试输入，而模拟器检测输出应答。简单的 2bit 比较器 testbench 如示例 1.7 所示。

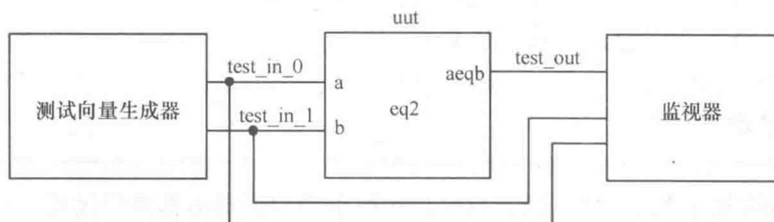


图 1-4　2bit 比较器 testbench

示例 1.7　2bit 比较器 testbench

```
// 时间精度指示仿真时间单位是1ns,时间步长为10ps
'timescale 1 ns/10 ps
module eq2-testbench;
  // 信号声明
  reg [1:0]test_in0,test_in1;
  wire test_out;
  // 测试底层电路示例
  eq2 uut
          (.a(test_in0),.b(test_in1),.aeqb(test_out));
// 向量生成程序
initial
begin
    // 测试向量1
    test_in0 = 2'b00;
    test_in1 = 2'b00;
    #200;
    // 测试向量2
    test_in0 = 2'b01;
    test_in1 = 2'b00;
    #200;
    // 测试向量3
    test_in0 = 2'b01;
    test_in1 = 2'b11;
    #200;
    // 测试向量4
    test_in0 = 2'b10;
    test_in1 = 2'b10;
    #200;
    // 测试向量5
    test_in0 = 2'b10;
    test_in1 = 2'b00;
    #200;
    // 测试向量6
```

```
    test_in0 = 2'b11;
    test_in1 = 2'b11;
    #200;
    // 测试向量7
    test_in0 = 2'b11;
    test_in1 = 2'b01;
    #200;
    // 结束仿真
    $ stop;
  end
endmodule
```

代码由 2bit 比较器一段模块例化语句和一个生成测试模式的次序的初始化块组成。初始化块是一种特殊的 Verilog 结构，在仿真开始时执行一次，一个初始化块内的声明语句从而执行。每一个测试模型由三条语句生成，如

```
// 测试向量2
test_in0 = 2'b01;
test_in1 = 2'b00;
#200;
```

前两条语句指定了 test_in0 和 test_in1 信号的值，而第三条指示两个值持续 200 个时间单元。最后的语句 $ stop 是一种用来停止仿真，返回到仿真软件控制的 Verilog 系统功能。

代码中没有监视器，我们能够在一种“虚拟逻辑分析仪”的仿真显示界面上观察输入和输出波形。testbench 的仿真定时矢量图如图 2-16 所示。

书写一段全面的测试向量发生和监视的代码需要详细的 Verilog 知识。对现在来讲，以上这段清单可以作为其他综合电路的 testbench 模板。我们可以通过新的电路来代替这段 uut 实例。在 7.5 节中，会提供有关附加的模型和仿真关联语言架构，同时示范一种更精确的 testbench 写法。

1.8　文献备注

在每一章的最后都有一段目录部分用于为未来研究提供一些相关的参考书目。而在本书的末尾包含一个全面的目录。

Verilog 是一种复杂的语言。在 S. Palnitkar 所著的《IEEE Standard Verilog Hardware Description Language，IEEE Std 1364-2011. Verilog HDL，2nd edition》一

书中详细说明了它的标准，而 M. D. Ciletti 所著的《Starter's Guide to Verilog 2001》中提供了详细的该语言的语法和结构。Verilog-2001 相对于旧的标准来说进行了许多改进。由 S. Sutherland 写的文章《The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need It》则概述其新的特征。编写一个大型数字系统的 testbench 是一项艰巨的任务。由 J. Bergeron 所写的《Writing Testbenches: Functional Verijication of HDL Models, 2nd edition》则以此为重点进行说明。

1.9 实验

在每一章的结尾部分都有一些练习用的实验。这些实验练习能够帮助我们更好地理解相关概念并提供设计和调试实际电路的实用机会。

1.9.1 编码

编写实验 2.9.1HDL 代码。在我们完成第 2 章的学习后，该段代码可以进行仿真和综合。

1.9.2 二进制解码器门级编码

编写实验 2.9.2HDL 代码。在我们完成第 2 章的学习后，该段代码可以进行仿真和综合。

第2章　FPGA及EDA软件概述

2.1　简介

开发一个大型的基于FPGA的系统是一个繁杂的过程，涉及许多复杂的转换和算法优化。因此需要一些软件工具使一些工作自动化。我们使用Web版的Xilinx ISE开发套件进行综合和实现，并使用Mentor Graphics ModelSim XE III入门版来仿真。本章中，我们对FPGA器件及S3开发板进行了简要的介绍，并通过提供两种软件包的简明教程，以方便读者快速进入学习过程。

2.2　FPGA

2.2.1　通用FPGA器件概述

现场可编程门阵列（FPGA）是一种由通用逻辑单元二维阵列和可编程开关组成的逻辑器件。FPGA器件的基本结构如图2-1所示。逻辑单元能够被配置（编程）以执行简单的功能，而可编程开关可以被定制，以提供对内部逻辑单元之间的互联。通过定义每个逻辑单元的功能并有选择地设置每个可编程开关即可实现一个定制的设计。完成了设计和综合以后，我们可以通过简单的适配器电缆将期望的逻辑单元和开关配置信息下载到FPGA器件中，以获得定制的电路。由于这一过程能够在“现场”完成，而不需要在制造设备中进行，因此这种器件被认为是现场可编程的。

基于LUT的逻辑单元　一个逻辑单元通常包含一个小的可配置组合电路及一个D触发器（DFF）。实现可配置的组合电路最通用的办法是使用查找表（LUT）。一个n输入的LUT可以看成一个$2^n \times 1$存储器。通过向存储器写入恰当的内容，我们便可以通过LUT实现任何一种n输入的组合逻辑功能。基于3输入LUT的逻辑单元结构如图2-2a所示。通过3输入LUT实现一个$a \oplus b \oplus c$功能的示例如图2-2b所示。注意，LUT的输出可以直接使用或存储于触发器中。后者可以用于实现时序电路。

宏单元　绝大多数FPGA器件嵌入了一定的宏单元或宏模块。这些是在晶体管级设计并生产的，其功能补充了普通的逻辑单元。通常使用的宏单元包括存储

器块、组合乘法器、时钟管理电路、I/O 接口电路。高级 FPGA 器件甚至会带有一个或多个预制的处理器核。

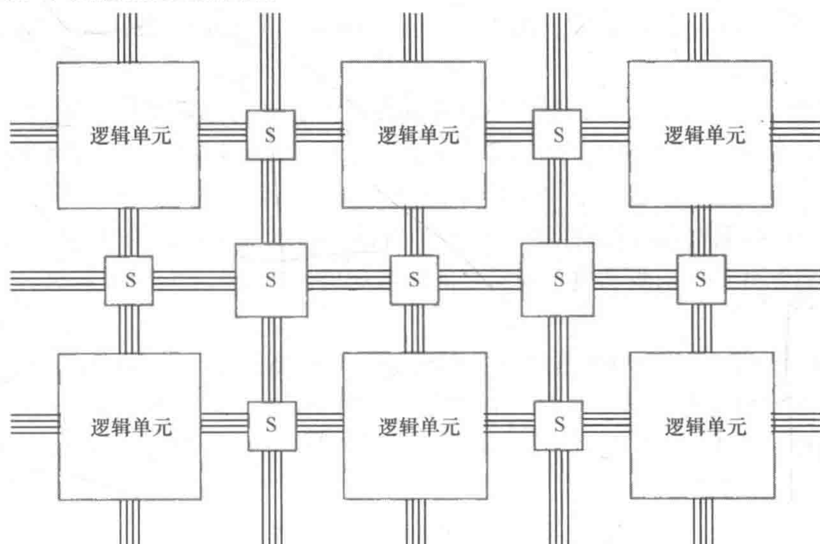
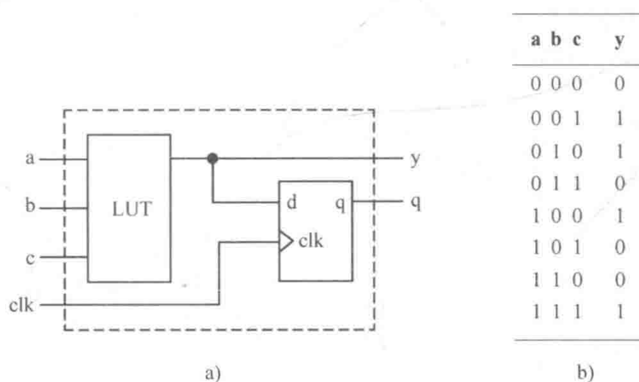


图 2-1　FPGA 器件基本结构



a b c	y
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

图 2-2　基于三输入 LUT 的逻辑单元

2.2.2　Xilinx Spartan-3 器件概述

本书使用 Xilinx Spartan-3 系列 FPGA 器件。基于逻辑单元和 I/O 数量的比率，该系列进一步分为几个子系列。我们的讨论适用于全部子系列。

逻辑单元、Slice 和 CLB　Spartan-3 器件中最基本的元件是逻辑单元（LC），它是由一个 4 输入查找表和一个 D 触发器组成，与图 2-2 所示的相似。另外，一个逻辑单元还会包含一个用于实现算数功能的进位电路和一个用于实现多路开关的多路开关电路。LUT 也可以配置为 1 个 16 × 1 的静态随机寻址存储器

（SRAM）或一个 16bit 的移位寄存器。

为了增加灵活性同时提高性能，8 个逻辑单元通过特殊的内部布线结构组合了起来。在 Xilinx 的术语中，两个逻辑单元组成一个 Slice，4 个 Slice 组成一个可配置逻辑块（CLB）。

宏单元　Spartan-3 系列器件包含 4 种类型的宏单元：组合乘法器、块 RAM、数字时钟管理器（DCM）和 I/O 块（IOB）。组合乘法器可对两个 18bit 数据进行乘积计算。块 RAM 是一个 18k bit 同步 SRAM，可以配置成多种不同类型的结构。DCM 使用数据延时环以减少时钟的偏移并且控制时钟的频率和相位偏移。IOB 控制器件 I/O 引脚和内部逻辑之间的数据流，通过配置可以支持多种 I/O 信号标准。

Spartan-3 子系列中的器件　尽管 Spartan-3 子系列 FPGA 器件中有相似的逻辑单元和宏单元类型，但它们的密度是不同的。每个子系列都包含一系列不同密度的器件。表 2-1 中汇总了 Spartan-3 子系列中各个器件中 LC、块 RAM、乘法器和 DCM 的数量。

表 2-1　Spartan-3 系列器件

型号	LC 数量	RAM 块数量	RAM 引脚模块	乘法器数量	DCM 数量
XC3S50	1,728	4	72kbit	4	2
XC3S200	4,320	12	216kbit	12	4
XC3S400	8,064	16	288kbit	16	4
XC3S1000	17,280	24	432kbit	24	4
XC3S1500	29,952	32	576kbit	32	4
XC3S2000	46,080	40	720kbit	40	4
XC3S4000	62,208	96	1,728kbit	96	4
XC3S5000	74,880	104	1,872kbit	104	4

2.3　Digilent S3 开发板概述

Digilent S3 开发板基于 Spartan-3 器件（通常为一个 XC3S200），并带有一系列固定的外设。简化的版图如图 2-3a 和 2-3b 所示。主要的元件和连接器如下。

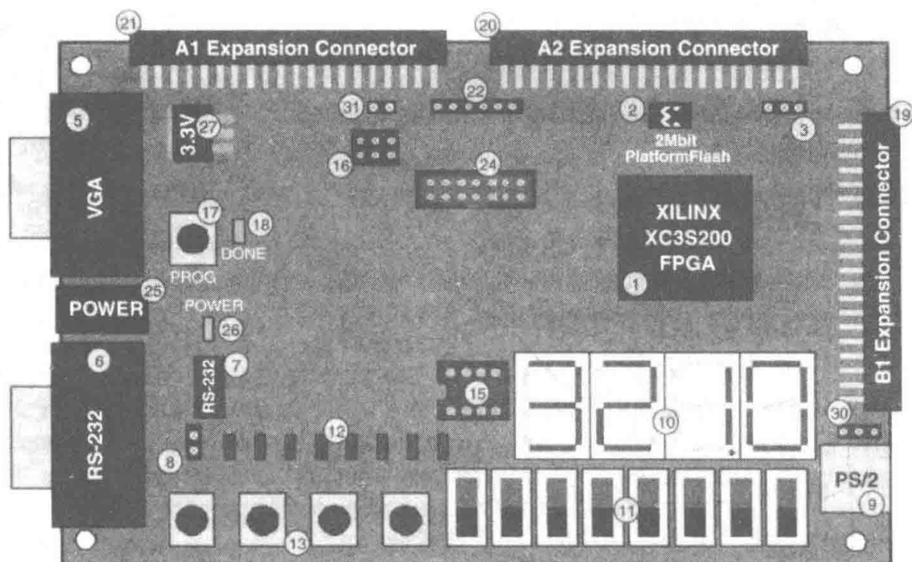
1）Xilinx Spartan-3 XC3S200 FPGA 器件（XC3S200FT256）；

2）2Mbit XILINX XCF02S platform Flash 配置 PROM；

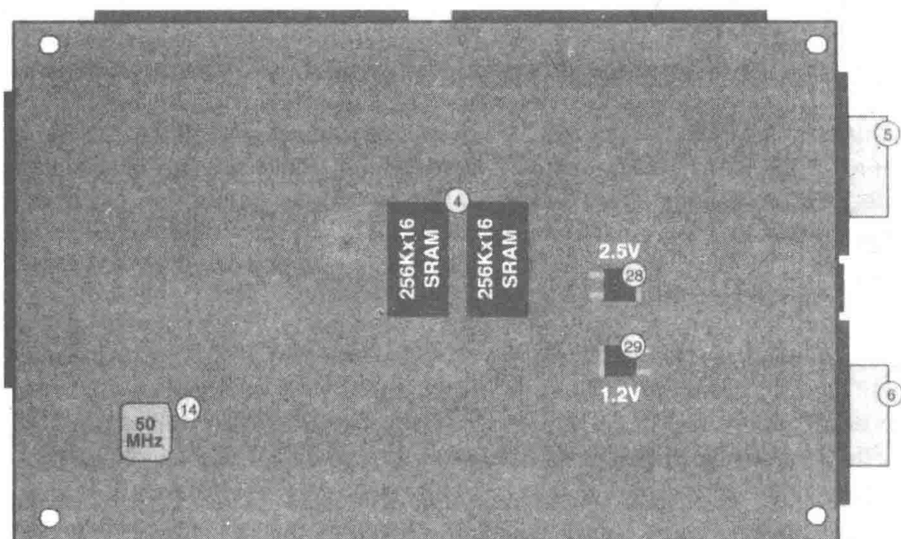
3）配置源选择跳线；

4）两块 256kbit × 16 异步 SRAM 器件（ISSI IS61LV25616AL-10T）；

5）VGA 显示端口；



a) 顶层视图



b) 底板视图

图 2-3　S3 开发板板图

6）RS-232 串口；

7）RS-232 收发器/电平转换器；

8）第二 RS-232 收发通道；

9）PS/2 鼠标/键盘接口；

10）4 位七段 LED 显示；

11）8 个拨动开关；

12）8 个独立的 LED 输出；

13）4 个按键开关；

14）50MHz 时钟晶振；

15）辅助时钟晶振插槽；

16）FPGA 配置模式选择跳线；

17）强制 FPGA 重新配置按键开关；

18）FPGA 配置成功指示 LED；

19）40 针扩展连接器 1（标识 B1）；

20）40 针扩展连接器 2（标识 A2）；

21）40 针扩展连接器 3（标识 A1）；

22）Digilent 下载电缆使用的 JTAG 连接器；

23）Digilent 低成本下载电缆（包括在 S3 套件中但未在图 2-3 中标识）；

24）JTAG 接口（用于 Xilinx Parallel Cable IV 和 MultiPRO Desktop Tool，不包括在 S3 套件中）；

25）非可调 5V 电源连接器（包括在 S3 套件中）；

26）供电指示 LED；

27）3.3V 稳压器；

28）2.5V 稳压器；

29）1.2V 稳压器；

30）PS2 端口供电电压（3.3V 或 5V）选择器。

2.4 开发流程

简化的 FPGA 系统开发流程如图 2-4 所示。为了便于阅读，我们遵循 Xilinx 文档中的术语。流程图的左半部分是优化和编程过程，在这一过程中，系统由抽象的 HDL 文本描述转换为器件单元级的配置数据，然后下载到 FPGA 器件中。流程图的右半部分是验证过程，检查设计是否满足功能和性能要求。流程中的主要步骤如下。

1）设计系统并形成 HDL 文件，我们可能需要增加一个单独的约束文件用来指定一定的实现约束；

2）用硬件描述语言开发 testbench 并实施 RTL 级仿真，RTL 是指 HDL 代码是在寄存器传输级设计完成的；

3）进行综合和实现，综合过程通常指逻辑综合，在这一过程中软件将 HDL

结构转换为普通的门级元件，例如简单逻辑门和触发器。实现过程包括 3 个更小的过程：转换、映射和布局布线，转换过程将多个设计文件合并到一个网表文件中，映射过程通常是指工艺映射，将网表中的通用的门转换为 FPGA 的逻辑单元和 IOB，布局布线过程通常是指通过布局和布线，得到 FPGA 芯片内部的物理版图，这一过程中将逻辑单元放置在具体的物理位置上并决定连接各种信号的走线，在 Xilinx 的流程中，在实现过程的最后要执行静态时序分析，静态时序分析得出各种时序参数，例如最大传播延时和最大时钟频率等；

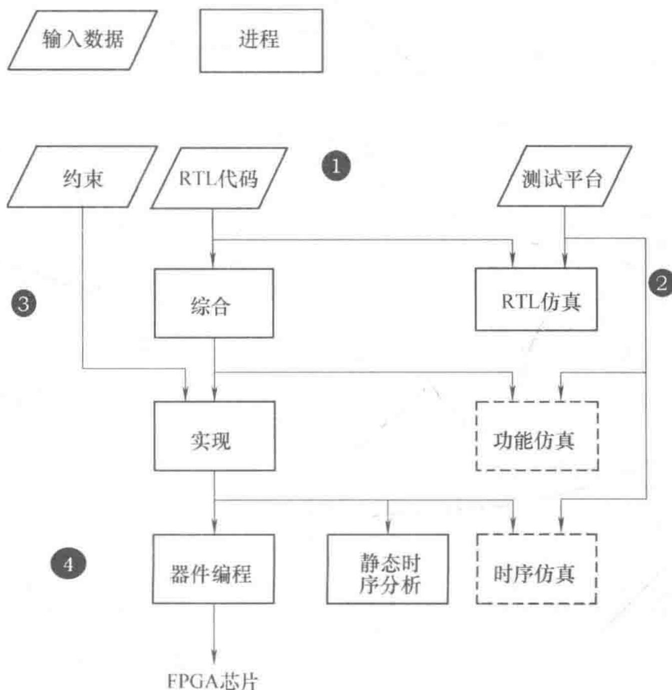


图 2-4　开发流程

4）生成并下载编程文件，在这一过程中会根据最终的网表生成配置文件，该文件被串行下载到 FPGA 器件中以配置逻辑单元和开关，相应的物理电路也会被验证。

门级功能仿真和时序仿真分别可在逻辑综合后和实现后实施，这两个流程是可选的。门级功能仿真使用综合后的网表替代 RTL 描述并可检查综合过程的正确性。时序仿真使用最终的布局布线后网表和详细的时序数据实施仿真。由于网表的复杂性，门级功能仿真和时序仿真需要消耗相当多的时间。如果我们遵循良好的设计和编码实践，HDL 代码将会被正确地综合和实现。我们只需要使用 RTL 仿真去检查 HDL 代码的正确性，并使用静态时序分析去检查相关时序信息

即可。门级功能仿真和时序仿真可以在开发流程中略去。

2.5　Xilinx ISE 图形化界面概貌

Xilinx ISE（集成软件环境）用于控制开发流程中的所有环节。Project Navigator 是一个图形化界面，用于用户访问软件工具和工程相关文件。除 ModelSim 仿真外，我们可以使用 ISE 启动所有开发任务。本节讨论的内容以及下一节的教程均基于 ISE WebPack 8.2 版本。

默认的 ISE 窗口如图 2-5 所示。它分为 4 个子窗口：



图 2-5　ISE 窗口

- Source 窗口（顶端左侧）：层次化显示工程中包含的文件；
- Processes 窗口（中间左侧）：显示对于当前选择的源文件能够进行的处理；
- Transcript 窗口（底端）：显示状态信息、错误和警告；
- Workplace 窗口（顶端右侧）：包含用于查看和编辑多个文档的窗口（例如 HDL 代码、报告、原理图等）。

每个子窗口可被缩放、移动、停靠或取消停靠。可以通过选择 VIEW→Restore 恢复默认布局。注意，一个子窗口可以包含多个页面。底下的标签用于选择想要的页面。

Source 窗口　Source 窗口主要用来显示与当前工程相关的文件。示例 2.2 中的设计是一个典型的 Source 窗口，如图 2-6 所示。顶部的下拉列表，标签为"Sources for:"，用于选择当前的设计视图。synthesis/implementation 视图只有在我们使用 ISE 进行综合和实现时才可被选择。

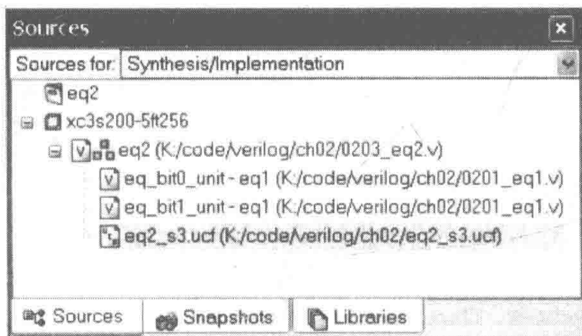


图 2-6　Sources 窗口

底部有 3 个标签，分别为 Source、Snapshots 和 libraries。Source 标签显示工程的名字、指定的 FPGA 器件、用户文档以及设计文件。模块会依据内部的设计层次显示。在图 2-6 中，eq2 和 eq1 实体反映了示例 2.2 的层次结构。eq2 模块中包含定义了设计约束的 eq_s3.ucf 文件。我们可以通过双击相应模块在 workplace 中打开文件。顶层模块图标被放在模块旁边，像 eq2 模块边上那样，用于调用对特定的模块的综合和实现过程。

Snapshots 标签显示了工程的快照，这些快照是以前存储的工程文件的副本。Libraries 标签显示了所有与工程相关的库。

Processes 窗口　Processes 窗口显示了可用的处理过程。该窗口是环境敏感的，具体来说，窗口中可以选择的处理过程与 Sources 窗口中选择的源文件类型有关。例如图 2-6 中，eq2 模块被设为了顶层模块并被选中，可以获得的处理过

程显示于 Processes 窗口中，如图 2-7 所示。一些处理过程可能还包含几个子处理过程。我们可以通过点击相应处理图标来开始一个处理过程。ISE 采用了自动处理技术，能够自动的运行必要的处理流程来达到预期的步骤。例如，当我们想启动 Generate Programming File 处理过程，ISE 会自动调用 Synthesize 和 Implement Design 处理过程，因为产生编程文件过程要使用实现过程的结果，而实现过程又要依靠综合过程的结果。

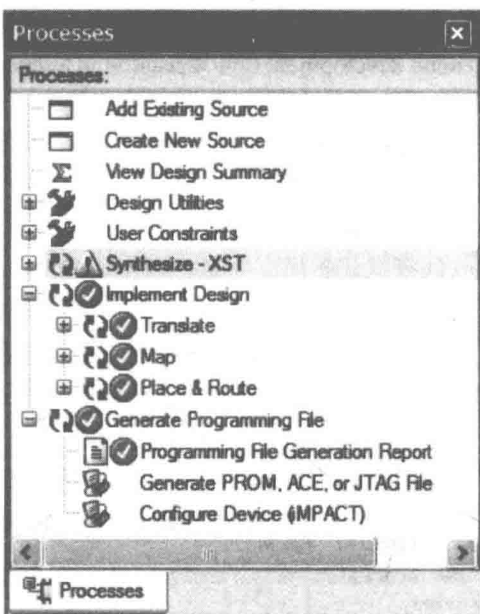


图 2-7　Processes 窗口

Transcript 窗口　Transcript 窗口用于显示处理过程的进展情况及相关信息。Console 页显示了错误、警告和通知消息。错误用红 X 标记在消息旁边，而警告用黄色！标记。Warnings 和 Errors 页只显示警告和错误消息。

Workplace 窗口　Workplace 窗口用于查看及编辑各种文件。通常我们只使用它完成两个主要工作。第一个工作是查看和编辑 HDL 及约束文件。默认的编辑器是 ISE Text Editor，这个简单的文本编辑器带有一些可以帮助编写 HDL 代码的特性。第二个工作是用于查看设计摘要和各种报告。

2.6　ISE Project Navigator 简明教程

Xilinx ISE 由一系列软件工具构成，然而对这些软件工具使用的详细讨论超出了本书的范围。这一章节我们只提供一段简明教程来阐明基本的开发过程。有 4 个主要步骤：

1）创建设计工程及 HDL 代码；

2）创建一个 testbench 并执行 RTL 级仿真；

3）添加约束文件并对代码进行综合和实现；

4）生成并下载配置文件至 FPGA 器件中。

以上步骤遵从 2.4 节中讨论的一般开发流程。在本教程中，我们使用在第 1 章讨论的 2-bit 比较器。代码如示例 2.1 和 2.2 所示。

示例2.1 1-bit比较器的门级实现

```
module eq1
  //I/O 端口
  (
   input wire i0, i1,
   output wire eq
  );
  // 信号声明
  wire p0, p1;
  // 程序体
  // 两乘积项之和
  assign eq = p0|p1;
  // 乘积项
  assign p0 = ~i0 & ~i1;
  assign p1 = i0 & i1;
```

示例2.2 2-bit比较器的结构描述

```
module eq2
  (
   input wire[1:0] a, b,
   output wire aeqb
  );
  // 内部信号声明
  wire e0, e1;
  // 程序体
  // 例化两个1bit 比较器
  eq1 eq_bit0_unit (.i0(a[0]), .i1(b[0]), .eq(e0));
  eq1 eq_bit1_unit (.eq(e1), .i0(a[1]), .i1(b[1]));
  // 若每位都相等则a 与b 相等
  assign aeqb = e0 & e1;
endmodule
```

2.6.1 创建工程和 HDL 代码

在这一步骤中，有 3 项任务：

- 创建一个工程；
- 添加或创建 HDL 文件；
- 检查 HDL 语法。

创建一个工程 ISE 工程包含一个设计的基本信息，包括源文件种类和目标器件。可以按如下步骤创建一个新工程。

1）选择 Start→All Programs→Xilinx ISE→Project Navigator（或者存放 ISE 的位置）来启动 ISE project navigator；

2）在 Project Navigator 中，选择 File→New Project，弹出 New Project Wizard - Create New project 对话窗，输入工程名称（如 eq2）及位置，并确认 Top-level Source Type 区域中选择的是 HDL，单击“Next”；

3）出现 The New Project Wizard-Device Properties dialog 对话框，我们需要选择该对话框中的目标器件，可在 FPGA 开发板手册中或 FPGA 芯片顶部的标记找到该信息，对于一个典型的 S3 开发板 ，选择如下：

- Product Category：All；
- Family：Spartan3；
- Device：XC3S200；
- Package：FT256；
- Speed：-4，我们还需要确认一下是否选择了 Xilinx XST 软件用于综合；
- Synthesis Tool：XST（VHDL/Verilog）；

4）多次单击“Next”按钮，完成余下的对话框，之后单击“Finish”来完成创建工程。

一个工程创建之后，我们可以创建或添加 HDL 文件和约束文件。

创建一个新的 HDL 文件 如果文件不存在，我们就必须创建一个新的源文件。创建一个新的 HDL 文件的过程如下。

1）选择 Project→New Source，出现 The New Source Wizard-Select Source Type dialog 对话框，选择 Verilog Module 并输入文件名称 eq2，单击“Next”；

2）出现下一个对话框，该对话框允许我们输入嵌入到 Verilog 代码中的端口名称，然而，由于自动生成的代码使用了旧的端口声明风格，因此我们不使用该特性，单击“Next”；

3）单击“Finish”，一个新的 HDL 文本编辑窗口便出现在 Workplace 窗口中，同时软件自动生成了一段注释标题和模块分隔符；

4）使用编辑器输入示例 2.2 中的 HDL 代码并保存文件；

5）重复以上步骤来创建另一个示例 2.1 中的文件。

添加已有文件　如果一个文件已经存在，则可以按照如下步骤添加到工程里。

1）选择 Project→Add Source，出现一个对话窗口；

2）转到适当的目录并选择目标文件，单击“Open”，出现一个新的对话框；

3）单击“OK”完成添加，这些文件此时便出现在了 Project Navigator 的 Sources 窗口中。

检查代码语法　完成一个新的 HDL 文件后，我们需要检查代码语法。

1）在 Source 窗口中选择目标文件；

2）在 Processes 窗口中，单击 Synthesize 附近的“+”图标展开处理层次；

3）双击“Check Syntax process”。

底部的 transcript 显示了处理的进展情况并报告错误和警告，分别以红色 X 和黄色！为开头标记。双击这些信息会转到文件中的错误行。我们可修改问题，保存文件，并重复语法检查过程直到消除所有语法错误。

2.6.2　创建 Testbench 及执行 RTL 仿真

Testbench 的功能就像一个虚拟的试验平台。它包括被测试的 HDL 模块和用来产生激励的代码。RTL 仿真在计算机中验证 HDL 模块的操作。ISE 包含一个内置的 ISE 仿真器，也可以调用 Mentor Graphics 公司的 ModelSim 仿真器。因为后者更加稳定和通用，因此本书中我们使用 ModelSim 仿真器。尽管 ModelSim 能够从 ISE Project Navigator 中调动，我们还是把它作为独立的软件工具，并在 2.7 节中进行说明。

2.6.3　添加约束文件综合和实现代码

这一步骤有 3 个任务：

- 添加约束文件；
- 执行综合和实现；
- 检查设计摘要。

添加约束文件　约束包括综合和实现过程的限制条件。主要的约束类型是对顶层 I/O 引脚的分配和最小时钟速率的设置。在实现过程中，顶层模块的 I/O 信号必须被映射到 FPGA 器件的物理引脚上。由于外围接口的 I/O 信号已经在开发板上与指定的 FPGA 引脚永久相连，我们必须保证信号映射到对应的引脚上。另一种约束的类型是关于时序的，它指定了最小时钟频率以便于使用板上的晶振。

约束信息存储在扩展名为 .ucf 的文本文件中（用户约束文件）。在 eq2 的电

路中，我们将 a 和 b 端口连接到 4 个开关上，aeqb 端口连到 LED 上以验证电路的物理操作。对于 S3 板，对应的引脚是 F12、G12、H14、H13 和 K12。约束文件为

```
# 4slideswitches
NET "a<O>" LOC = "F12" ; # switch0
NET "a<l>" LOC = "G12" ; # switch1
NET "b<O>" LOC = "H14" ; # switch2
NET "b<l>" LOC = "H13" ; # switch3
# led
NET "aeqb" LOC = "K12" ; # led0
```

#用于注释，在该符号之后的文本是可被忽略的。这个文件必须在 Source 窗口添加到设计中。

有一些 ISE 工具可以用来设置和产生约束文件。由于我们的实验都是在相同的原型板中做的，因此约束（例如引脚分配和时钟频率）也保持不变。附录中提供了包含 S3 板中所有连接的外围接口信号的约束文件模板。创建约束文件的一个简单的办法是根据当前设计的 I/O 端口名称复制和编辑模板。为 eq2 电路创建 . ucf 文件的过程如下。

1）复制模板中的约束文件并重命名为 eq2_s3. ucf；

2）在 Source 窗口中按照 2. 6. 1 节中的过程为 eq2 模块添加新的约束文件；

3）选定约束文件；

4）在 Processes 窗口中，单击 User Constraints 旁边的“+”图标，展开处理的层次；

5）双击“Edit Constrains（Text）”过程启动 ISE 文本编辑器；

6）根据需要重命名 I/O 名字，然后删除无用的引脚分配；

7）保存文件。

ISE 8. 2 版本的默认选项只允许对已有的顶层 I/O 端口进行引脚分配。如果没有从 ucf 模板中删除无用的引脚分配，则会产生错误提示。我们可以用如下方式覆盖默认的选项。

1）选择顶层 HDL 文件；

2）在 Processes 窗口中鼠标右键单击“Implement Design”过程，在菜单中选择“Properties...”，会出现一个对话框；

3）在对话框中在“Allow Unmatched LOC Constraints”选项处打勾，单击“OK”。

当这个选项打开后，我们可以使用相同的 ucf 模板用于所有的设计，只要顶层模块中的 I/O 端口名称是相同的，无需每次编辑 ucf 文件。

执行综合和实现 调用综合和实现过程非常简单：

1）选择要被综合的模块并确认它被指定为设计的顶层模块（在该模块图标旁带有一个绿色的方块）；

2）在 Processes 窗口中双击“Implement Design”过程；

3）尽管之前进行过语法检查，代码中仍可能包含一些无法被综合或不利于实现的结构（例如组合逻辑反馈环），错误和警告信息被显示在 transcript 窗口中的 console 标签内；

4）更正问题并重复仿真和综合的过程。

检查设计摘要 随着工程的推进，在每个过程中都会产生报告。这些报告和关键的统计数据会显示在 design summary 窗口中。我们可以检查形成的电路大小（包括 slices 数、触发器个数和查找表个数）。对于时序电路，检查时钟频率是否满足时序约束。该摘要可以通过在 Processes 窗口中双击“View Design Summary”过程来查看。eq2 电路的设计摘要如图 2-8 所示。我们可以在“Device Utilization Summary”部分查看 slice、LUT 等资源的使用。更详细的报告可以通过点击相应的链接来查看。

2.6.4 生成并下载配置文件至 FPGA 芯片

最后一步是生成配置文件并下载到 FPGA 器件中。该步骤中有 3 个任务：

- 连接下载线；
- 产生配置文件；
- 下载配置文件；

S3 开发套件带有一个并口 JTAG 下载线，后面的讨论就是基于此下载线。其他下载线的使用过程与之类似，在使用手册中有详细的说明。

连接下载线 将开发板做如下准备：

1）确认 PROM 和配置模式跳线（图 2-3 中标识为 3 和 16）为默认设置；

2）连接电源线；

3）连接下载线的一端到 PC 的并口，连接另一端到 S3 板的 JTAG 接口（图 2-3 标识为 22）。

生成配置文件 生成配置文件非常简单：

1）确认在 Source 窗口中选中顶层模块；

2）在 Processes 窗口中点击“Generate Programming File”。

完成这步后配置文件 eq2. bit 便产生了。

下载配置文件 下载配置文件到 FPGA 器件中是通过软件工具 iMPACT 完成的，可以从 ISE Project Navigator 中调用。过程如下。

1）在 Processes 窗口单击“ + ”展开“Generate Programming File”过程；

EQ2 Project Status			
Project File:	eq2.ise	**Current State:**	Placed and Routed
Module Name:	eq2	• **Errors:**	No Errors
Target Device:	xc3s200-5ft256	• **Warnings:**	No Warnings
Product Version:	ISE, 8.1i	• **Updated:**	Sun Jan 21 18:04:45 2007

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	1	3,840	1%	
Logic Distribution				
Number of occupied Slices	1	1,920	1%	
Number of Slices containing only related logic	1	1	100%	
Number of Slices containing unrelated logic	0	1	0%	
Total Number of 4 input LUTs	1	3,840	1%	
Number of bonded IOBs	5	173	2%	
Total equivalent gate count for design	6			
Additional JTAG gate count for IOBs	240			

Performance Summary			
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints		

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sat Jan 20 22:22:32 2007	0	0	0
Translation Report	Current	Sat Jan 20 22:22:46 2007	0	0	0
Map Report	Current	Sat Jan 20 22:23:00 2007	0	0	2 Infos
Place and Route Report	Current	Sat Jan 20 22:23:18 2007	0	0	1 Info
Static Timing Report	Current	Sat Jan 20 22:23:30 2007	0	0	2 Infos
Bitgen Report					

图 2-8 设计摘要

2）双击“Configure Device（IMPACT）”过程，出现 iMPACT 的欢迎界面，如图 2-9 所示，使用边界扫描（JTAG）检查配置器件，并核实已自动连接至下载线，同时确认在下拉列表中已选择 Automatically connect to a cable and identify Boundary-Scan chain 选项，单击“Finish”；

3）如果消息显示发现两个器件，则单击“OK”继续；

4）iMPACT 主窗口和“Assign New Configuration File”对话框同时出现，如图 2-10 所示，板子上连接到 JTAG 链的器件应被检测并显示出来；

5）选择 eq2. bit 文件并单击“Open”，将配置文件指派到 JTAG 链上的 xc3s200 上；



图 2-9　iMPACT 欢迎对话框

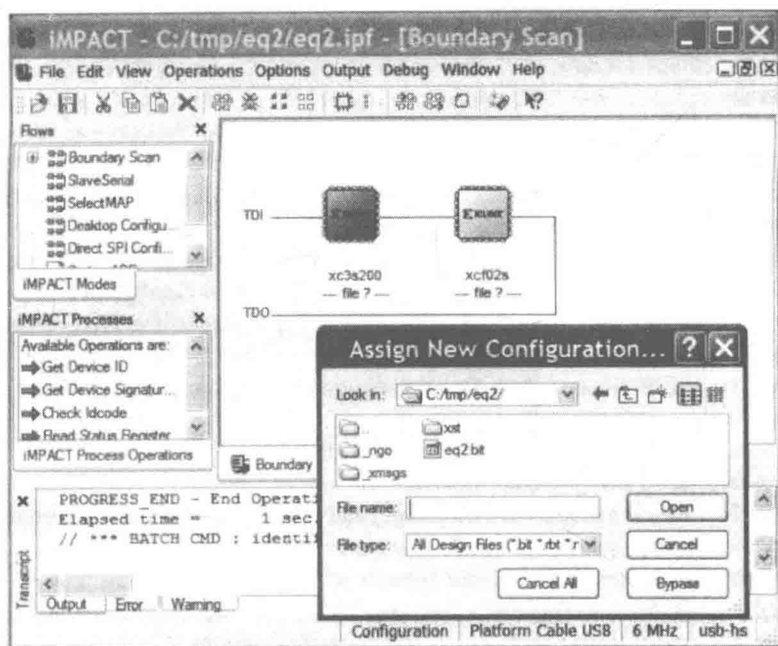


图 2-10　iMPACT 主窗口

6）如果出现警告信息，则忽略并单击“OK”；

7）选择 Bypass 来跳过其他器件；

8）右击 xc3s200 器件映像，选择“Program...”，Programming Properties 对话框被打开，单击“OK”对器件进行编程；

9）当下载过程完成时会出现 Program Succeeded 信息。

现在 FPGA 器件已经被配置，我们可以通过开关来测试电路，观察 LED 输出。

另外一种配置 FPGA 的方式是将配置文件下载到 PROM 中并且从 PROM 中加载配置文件。更多信息可以在文献备注章节所引用的资料中找到。

2.7　Modelsim HDL 仿真器简明教程

ModelSim 软件是 Mentor Graphics 公司的 HDL 仿真器，能够独立于 ISE 运行。本节的讨论基于 ModelSim XE Ⅲ 6.0d 入门版。

默认的 ModelSim 窗口如图 2-11 所示，它分为 3 个子窗口：Transcript 窗口（下部），Workspace 窗口和多文档界面（MDI）。Workspace 窗口显示当前处理的信息。底部的标签用于选择想要的过程页面，可以选择 Project、Library、Sim 等。

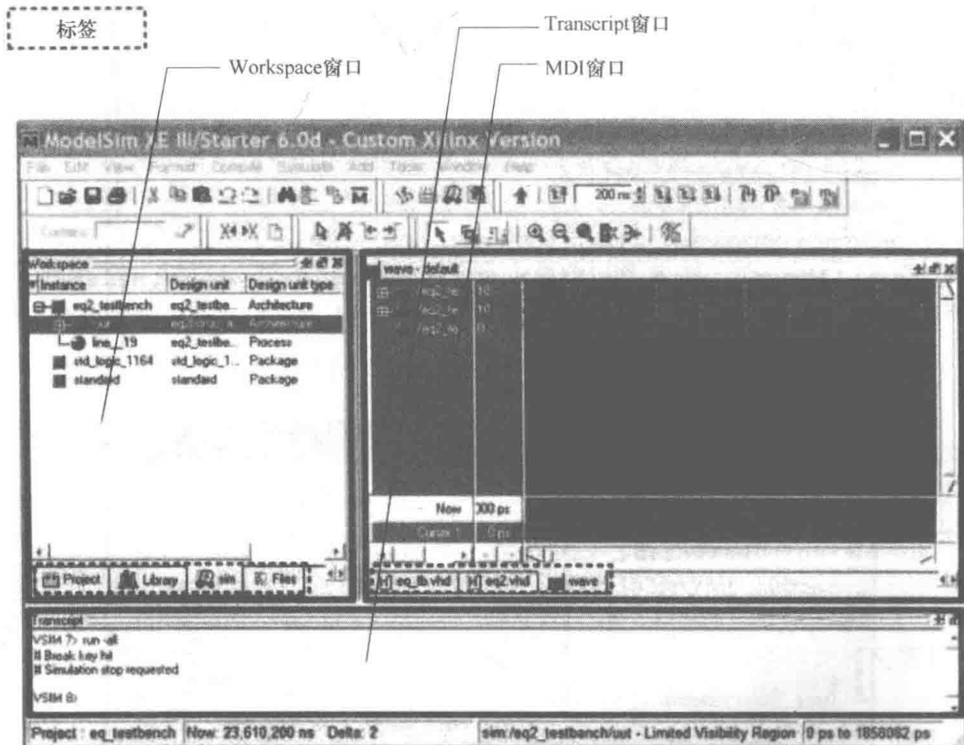


图 2-11　典型的 ModelSim 窗口

Transcript 窗口用于跟踪命令历史记录和信息，也可以作为命令行接口输入 ModelSim 的命令。MDI 窗口用于显示 HDL 文本、波形等。下面的标签用于选择想要的页面。

每个子窗口可以缩放、移动、停靠或取消停靠。另外一些窗口可以用于一些其他操作。可以通过选择 Window→Initial Layout 来恢复默认界面。

本节中，我们通过给出简明教程来说明基本的仿真过程，包括 3 步：

1）准备仿真工程；

2）编译 HDL 代码；

3）执行仿真并检查波形。

我们使用第 1 章的 2 输入比较器，代码在示例 2.3 中给出。

示例 2.3　2bit 比较器的 testbench

```
// 时间精度指示仿真时间单位是1ns ,时间步长为10ps
'timescale 1 ns/10 ps
module eq2_testbench;
  // 信号声明
  reg [1:0] test_in0, test_in1;
  wire test_out;
  // 测试底层电路示例
  eq2 uut
    (.a(test_in0), .b(test_in1), .aeqb(test_out));
  // 向量生成程序
  initial
  begin
    // 测试向量1
    test_in0 =2'b00;
    test_in1 =2'b00;
    #200;
    // 测试向量2
    test_in0 =2'b01;
    test_in1 =2'b00;
    #200;
    // 测试向量3
    test_in0 =2'b01;
    test_in1 =2'b11;
```

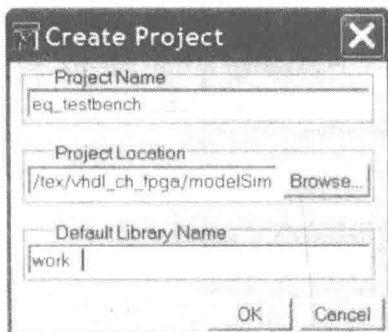
```
    #200;
    // 测试向量4
    test_in0 =2'b10;
    test_in1 =2'b10;
    #200;
    // 测试向量5
    test_in0 =2'b10;
    test_in1 =2'b00;
    #200;
    // 测试向量6
    test_in0 =2'b11;
    test_in1 =2'b11;
    #200;
    // 测试向量7
    test_in0 =2'b11;
    test_in1 =2'b01;
    #200;
    // 结束仿真
     $ stop;
  end
endmodule
```

准备仿真工程　ModelSim 仿真工程包括仿真库和 HDL 文件。Testbench 是一种 HDL 程序，它能够像我们在 2.6.1 节中讨论的那样使用 ISE 文本编辑器创建。另外，ModelSim 也自带了编辑器。我们假设所有的 HDL 文件都已构建完毕。创建一个工程的过程如下。

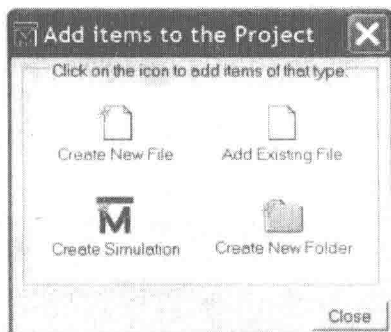
1）选择开始→所有程序→ModelSim XE Ⅲ 6.0d→ModelSim（或者 ModelSim 存放的地方）启动 ModelSim 程序；

2）选择 File→New→Project，出现创建工程对话框 Create Project，如图 2-12a 所示，输入工程名 eq_ testbench，选择工程位置，设置 Default Library Name 为 work，单击“OK”，在主窗口中出现一个空的工程页面，出现添加项目至工程窗口，如图 2-12b 所示。

3）在添加项目至工程窗口中单击“Add Existing File”，添加所需的 HDL 文件，单击“OK”，在 Workplace 子窗口中出现工程标签并显示出已选文件，如图 2-13 所示。



a) 创建工程对话框



b) 添加项对话框

图 2-12　新工程对话框

编译 HDL 代码　这里的编译是指将 HDL 代码转换为 ModelSim 内部格式。以 Verilog 为例，编译以模块为基础。其过程如下。

1）高亮显示 eq1 文件并右击鼠标，选择 Compile→Compile Selected，注意编译应从设计的底层模块开始，编译过程和信息在 transcript 窗口中显示；

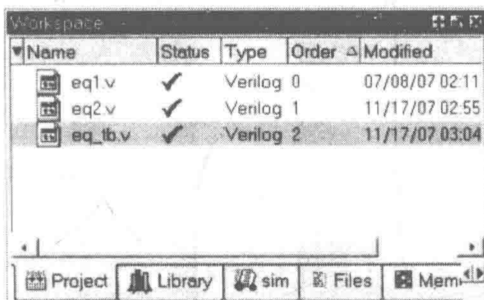


图 2-13　Workplace 面板的 Project 标签

2）如果文件没有语法错误，则会显示一个对勾，否则会显示 X，单击 transcrit 窗口中的红色错误行可定位错误，修正问题，保存文件并重新编译文件；

3）重复上述过程步骤，将 eq2 文件和 eq_tb 文件编译。

执行仿真并检查波形　在编译好 testbench 和相应文件后，我们可以执行仿真并检查结果波形。这相当于在虚拟实验环境下运行电路并在虚拟逻辑分析仪中检查波形。过程如下：

1）选择 Simulate→Start Simulation，出现仿真窗口；

2）在 Design 标签中，找到我们创建的 work 库，并展开，会显示出所有编译的单元，如图 2-14 所示。

3）通过双击相应的图标载入 eq2-testbench，在工作区窗口中出现 sim 标签并显示 eq2-testbench 的结构，如图 2-15 所示，object 窗口中会显示选中模块所包含的信号；

4）选中 uut 单元并右击鼠标，选择 Add→Add to Wave，会把所有 uut 单元的信号都加入到波形页面，波形页面会出现在 MDI 窗口中；

5）如果有必要，则重新安排信号的顺序并设置适当的格式（十进制、十六进制等）；

6）选择 Simulate→Run，有几个命令用来控制仿真：Restart（重置仿真），Run（运行一步仿真），Continue run（从中断中恢复运行），Run All（连续运行仿真），Break（中断仿真），这些命令在顶层窗口中也会以图标方式显示；

7）波形窗口显示仿真结果，如图 2-16 所示。我们可以滚动窗口，放大、缩小检查设计的正确性。

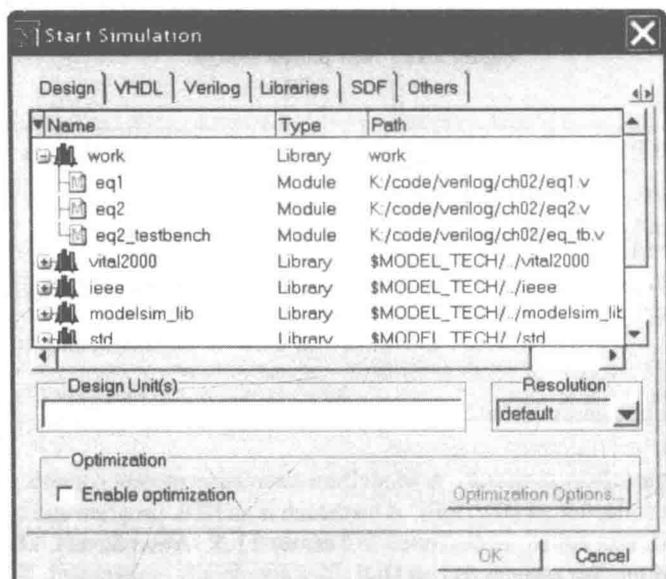


图 2-14　仿真对话框

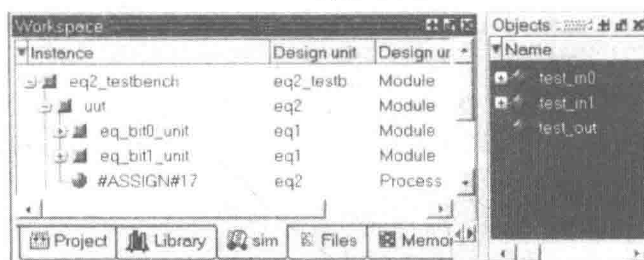


图 2-15　Workplace 的仿真面板

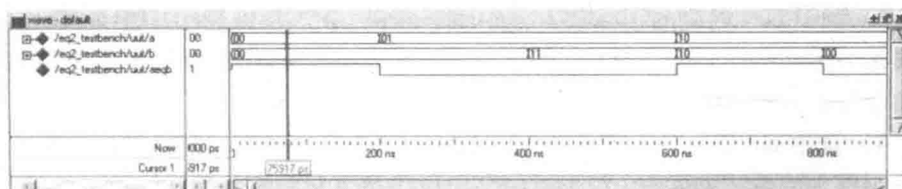


图 2-16　波形窗口

2.8　文献备注

Xilinx ISE 和 Mentor Graphics ModelSim 都是复杂的软件包，它们的文档超过数千页。大多数的文档都可以通过 Help 菜单访问。ISE 有一个 30 页的简明教程《ISE 8.1i Quick Start Tutorial》，更详细的 170 页的教程是《ISE In-Depth Tutorial》。Modelsim 也有类似的教程 ModelSim Tutorial。这些教程提供了关于软件包的所有特点的概述。Spartan-3 器件的相关信息可以从它的数据手册 DS099《Spartan-3 FPGA Family：Complete Data Sheet》中找到，包含了对逻辑单元和宏单元的详细解释。Clive Maxfield 的《Design Warrior's Guide to FPGAs》对 FPGA 的相关问题进行了全面的概述。S3 开发板的详细板图和 I/O 连接器可以在《Spartan-3 Starter Kit Board User Guide》中找到。其他开发板的相关信息可以在其手册中找到。

2.9　实验

2.9.1　门级大于电路

大于电路比较两个输入 a 和 b，当 a 大于 b 时，置输出有效。我们希望仅用门级逻辑运算符自底向上创建一个 4bit 大于电路。按如下步骤设计电路。

1）推导出 2bit 大于电路的真值表，并得到乘加形式的逻辑表达式，基于该表达式，只使用逻辑运算形成 HDL 代码；

2）为 2bit 大于电路设计 testbench，执行仿真并验证设计的正确性；

3）使用 4 个开关作为输入，1 个 LED 作为输出。综合电路并下载配置文件到开发板中，验证电路的运行；

4）使用 2bit 大于电路和 2bit 等于比较器以及最少的“胶连门”来构建 4bit 大于电路，先画出框图并根据框图得出结构级 HDL 代码；

5）为 4bit 大于电路设计 testbench，执行仿真并验证设计的正确性；

6）使用 8 个开关作为输入，1 个 LED 作为输出，综合电路并下载配置文件到开发板中，验证其功能。

2.9.2　门级二进制译码器

一个 n:2^n 二进制译码器可根据输入组合设定 2^n 位输出。一个带使能信号的 2:4 译码器的真值表见表 2-2。我们想使用门级逻辑运算符来创建译码器。过程如下：

1）确定带使能的 2:4 译码器的逻辑表达式，使用逻辑运算符形成 HDL 代码；

2）为译码器设计 testbench，执行仿真并验证设计的正确性；

3）使用两个开关作为输入，4 个 LED 作为输出，综合电路并下载配置文件至开发板中，验证其功能；

4）用 2:4 译码器推导出 3:8 译码器，首先画出框图并根据框图得到结构 HDL 代码；

5）为 3：8 译码器设计 testbench，执行仿真并验证设计的正确性；

6）使用 3 个开关作为输入，8 个 LED 作为输出，综合电路并下载配置文件至原型板中，验证其功能；

7）用 2:4 译码器推导出 4:16 译码器，首先画出框图并根据框图得到结构级 HDL 代码；

8）为 4:16 译码器设计 testbench，执行仿真并验证设计的正确性。

表 2-2 带使能 2:4 译码器真值表

输入			输出			
使能	a(1)	a(0)	b 代码			
1	—	—	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

第 3 章　寄存器传输级组合逻辑电路

3.1　引言

第 1 章介绍了利用简单的按位运算符来描述门级电路设计，门级电路均由简单的逻辑单元构成。本章中，我们将考查用 HDL 描述由中规模元件组成的电路，如加法器、比较器和乘法器等。由于这些组件是寄存器传输方法学中使用的基本构成块，因此很多时候称之为 RTL 级设计。我们会介绍更复杂一些的 Verilog 运算符、always 块、布线结构，并通过一系列的示例说明 RTL 级组合逻辑电路的设计。

3.2　运算符

Verilog 包含约 24 个运算符，见表 3-1。除在第 1 章中介绍的按位运算符，还有算术、移位和关系运算符。这些运算符对应中规模元件，如加法器和比较器。我们将在这一章节介绍这些运算符，同时还包括多种与综合相关的 Verilog 结构。

表 3-1　Verilog 运算符汇总

运算符类型	运算符标识	说　明	操 作 目 数
算术运算符	+	加法	2
	-	减法	2
	*	乘法	2
	/	除法	2
	%	取模	2
	**	求幂	2
移位运算符	>>	逻辑右移	2
	<<	逻辑左移	2
	>>>	算术右移	2
	<<<	算术左移	2

（续）

<table>
<tr><th>运算符类型</th><th>运算符标识</th><th>说　明</th><th>操作目数</th></tr>
<tr><td rowspan="4">关系运算符</td><td>></td><td>大于</td><td>2</td></tr>
<tr><td><</td><td>小于</td><td>2</td></tr>
<tr><td>>=</td><td>大于等于</td><td>2</td></tr>
<tr><td><=</td><td>小于等于</td><td>2</td></tr>
<tr><td rowspan="4">相等运算符</td><td>==</td><td>逻辑相等</td><td>2</td></tr>
<tr><td>!=</td><td>逻辑不等</td><td>2</td></tr>
<tr><td>===</td><td>全等</td><td>2</td></tr>
<tr><td>!==</td><td>全不等</td><td>2</td></tr>
<tr><td rowspan="4">按位运算符</td><td>~</td><td>按位非</td><td>1</td></tr>
<tr><td>&</td><td>按位与</td><td>2</td></tr>
<tr><td>|</td><td>按位或</td><td>2</td></tr>
<tr><td>^</td><td>按位异或</td><td>2</td></tr>
<tr><td rowspan="3">规约运算符</td><td>&</td><td>归约与</td><td>1</td></tr>
<tr><td>|</td><td>归约或</td><td>1</td></tr>
<tr><td>^</td><td>归约异或</td><td>1</td></tr>
<tr><td rowspan="3">逻辑运算符</td><td>!</td><td>逻辑非</td><td>1</td></tr>
<tr><td>&&</td><td>逻辑与</td><td>2</td></tr>
<tr><td>||</td><td>逻辑或</td><td>2</td></tr>
<tr><td rowspan="2">连接运算符</td><td>{ }</td><td>连接</td><td>任意</td></tr>
<tr><td>{ { } }</td><td>复制</td><td>任意</td></tr>
<tr><td>条件运算符</td><td>? :</td><td>条件</td><td>3</td></tr>
</table>

3.2.1　算术运算符

算术运算符共包括 6 个：+、-、*、/、% 和 **，分别为加、减、乘、除、取模及求幂操作。加、减运算符同时也可用于单目操作中，如 -a；在综合过程中，+、- 运算符会被推断为加法器和减法器，而这些加减法器是由 FPGA 内部的基本逻辑单元综合而成。

乘法是一种复杂的操作，对乘法运算符 * 的综合取决于综合工具软件和目标器件工艺。

Xilinx Spartan-3 FPGA 包含预制的组合逻辑乘法器模块。Xilinx 的综合工具软件 XST 能在综合过程中推断出乘法模块，这样，乘法运算符可以应用于 HDL 代码中。S3 开发板上面的 XCS200 FPGA 包含了 12 个 18×18 的乘法单元。虽然，乘法运算

符是支持综合的，但我们在使用时仍要注意乘法模块输入数据的位宽限制。

/、% 和 ** 运算符通常情况下不能被自动综合。

3.2.2 移位运算符

移位运算符包括4个：>>、<<、>>>和<<<，前2个为向右和向左的逻辑移位运算符，后2个为向右和向左的算术移位运算符。

当使用逻辑移位运算符（即>>、<<）时，0被移入。当使用>>>时，符号位（即最高位）被移入，当使用<<<运算符时，0被移入。注意：<<与<<<无区别，后者包括所有位。一些移位操作的例子详见表3-2。

表3-2 移位操作的例子

a	a >>2	a >>>2	a <<2	a <<<2
0100_1111	0001_0011	0001_0011	0011_1100	0011_1100
1100_1111	0011_0011	1111_0011	0011_1100	0011_1100

如果移位运算符的操作数均为信号，如a <<b，则该操作会被推断为桶式移位器，这是个相当复杂的电路。另一方面，如果移位数目是确定的，如a <<2，对该操作的综合只会涉及对输入信号的布线，而不会推断出任何逻辑。

该类型的操作也可通过3.2.5节中的catenation运算符描述。

3.2.3 关系和等价运算符

关系运算符包括>、<、≤和≥。这些运算符对2个操作数进行比较并返回一个布尔型结果，可能是true（由1bit标量1表示）或false（由1bit标量0表示）。

等价运算符包括4个：==、!=、===和!==，和关系运算符一样，等价运算符也返回true（1bit1）或false（1bit0）。===和!==运算符，分别称为case等式运算符和case不等式运算符，这两种运算符会对X和Z的值也进行比较，两者不能被综合。

关系运算符==和!=在综合时会生成比较器。

3.2.4 按位运算符、缩减运算符和逻辑运算符

按位运算符、缩减运算符和逻辑运算符、与、或、异或及非运算类似，由基本的逻辑单元实现。

1. 按位运算符 按位运算符包括四种：&（与）、|（或）、^（异或）和~（非）操作，前3个运算符需要2个操作数。非运算和异或运算，可以组合成异或非运算。由于这些运算符是以比特为运算单位，因此为按位运算符。例如，假

设 a，b，c 为 4bit 信号：

```
wire [3:0] a,b,c ;
语句
assign c = a | b ;
等同于：
assign c[3] = a[3] | b[3];
assign c[2] = a[2] | b[2];
assign c[1] = a[1] | b[1];
assign c[0] = a[0] | b[0];
```

2. 缩减运算符　& （与）、| （或）和 ~ （非）操作可以只有 1 个操作数，此时这三种运算符被称为缩减运算符。操作数一般为向量，指定的操作会对向量中的所有位进行运算并返回一个 1bit 的结果。例如，假设 a 为 4bit 信号，y 为 1bit 信号：

```
wire[3:0] a;
wire y;
语句
assign y = |a;// 单操作数
等同于
assign y = a[3]|a[2]|a[1]|a[0];
```

3. 逻辑运算符　逻辑运算符包括：&& （逻辑与）、|| （逻辑或）和 ! （逻辑非）。逻辑运算符与位运算符不同。假设不使用 X 或 Z，逻辑运算符的操作数将被识别为假（所有位为 0）或真（至少一位为 1），返回结果也为 1 位的结果。逻辑运算符可被理解为布尔表达式的逻辑连接，见表 3-3，相应的位运算符也在表中，用于说明两种运算符的差别。由于 Verilog 采用 0 和 1 表达假和真，位运算符和逻辑运算符在某些情况下可以通用。然而，还是建议在布尔表达式中使用逻辑运算符，在信号操作中使用位运算符。

表 3-3　逻辑及位操作示例

a	b	a&b	a|b	a&&b	a||b
0	1	0	1	0 (false)	1 (true)
000	000	000	000	0 (false)	0 (false)
000	001	000	001	0 (false)	1 (true)
011	001	001	011	1 (true)	1 (true)

3.2.5　位拼接和复制运算符

位拼接运算符 {}，用于将两个或多个信号的某些位拼接起来。下面的例子

说明了具体使用方法:

```
wire a1;
wire [3:0] a4;
wire [7:0] b8, c8, d8;
...
assign b8 = {a4, a4};
assign c8 = {a1, a1, a4, 2'b00};
assign d8 = {b8[3:0], c8[3:0]};
```

位拼接运算符的实现涉及输入信号和输出信号的重新连接，只需要 "wiring" 位拼接符的一个应用是对某信号进行确定位数的移位或者循环移位，例如

```
wire [7:0] a;
wire [7:0] rot,shl,sha;
// 对a 进行右循环移位3 位
assign rot = {a[2:0], a[8:3]};
// 对a 进行右循环移位3 位,并插入0(逻辑移位)
assign shl = {3'b000,a[8:3]};
// 对a 进行右循环移位3 位,并插入最高位(算术移位)
assign sha = {a[8], a[8], a[8], a[8:3]};
```

复制运算符 N{}，用于复制括号内的字符,并进行拼接。常数 N 表示复制的次数。例如，{4{2'b01}}返回 8'b01010101。

上面的算术移位也可简化为 assign sha = {3{a[8]},a[8:3]};

3.2.6 条件运算符

条件运算符?:，包含 3 个操作数，一般形式为

```
[signal] = [boolean - exp]? [true-exp]:[false-exp];
```

[boolean-exp]为布尔型表达式,返回 true(1'b1)或 false(1'b0)。当[boolean-exp]的值为真时,[signal]的值为[true-exp],否则为[false-exp]。例如,下面的电路获取了 a 和 b 中的最大值:

```
assign max = (a > b)? a:b;
```

该运算符可看作是简化的 if-then-else 语句:

```
if [boolean-exp] then
[signal] = [true-exp];
else
[signal] = [false-exp];
```

抛开其简洁性，条件运算符也可通过级联或嵌套的方式列举出期望的选择。

例如，表 1-1 中的等式电路，可通过条件运算符表达如下：

```
assign eq = ( ~il & ~i0)? 1'b1:
( ~i1 & i0)? 1'b0:
(i1 & ~i0)? 1'b0:
1'b1;
```

同样，我们可以通过扩展最大值比较电路，用来返回 a、b 和 c 的最大值：

```
assign max = (a > b)? ((a > c)? a:c):
((b > c)? b:c);
```

综合时，条件运算符会被推断为 2 选 1 的多路选择电路，电路详见 3.6 节。

3.2.7　运算符优先级

运算符优先级是指运算的顺序，优先级见表 3-4。计算表达式时，最高优先级的运算符最先被运算。如，表达式 a + b >>1，a + b 首先被计算，然后计算 >>1。可以通过插入括号来改变计算顺序，如 a + （b >>1）。我们推荐使用括号，使表达式更清晰，如（a + b） >>1，虽然对 a + b 插入括号不是必需的。

表 3-4　运算符优先级

运　算　符	优　先　级
! ~ + -（一元）	最高
** */% + -（二元）	
>> << >>> <<<	
<<=>>= ==!====!==	
& ^ |	
&& ||	
?:	最低

3.2.8　表达式位长度调整

正如真实硬件中的信号一样，Verilog 程序中的线和变量具有不同的位宽（即位长度或宽度）。在 Verilog 语句中，操作数的位宽是允许不同的，但位宽的调整需遵循以下规则；

- 在上下文中确定操作数的最大位宽，包括右侧的表达式和左侧的信号；
- 扩展右侧操作数的位宽至最大值并计算表达式的值；
- 将结果赋给左侧的信号，如果信号位宽较小则截去高位。

首先看一个简单的例子：

```
wire [7:0] a, b;
assign a = 8'b00000000;
```

```
assign b =0;
```

第一个赋值语句将8位信号“00000000”赋值给a。第二个赋值语句将整型数0赋值给b。由于Verilog中的整型数为32位，因此0被表示为“00000000000000000000000000000000”。由于b为8位宽度，因此赋值时会将b的值截断为“00000000”。虽然两句都是将全0赋值给信号，但我们应注意最后的值是如何得到的。

再看另外一个例子：

```
wire [7:0] a, b;
wire [7:0] sum8;
wire [8:0] sum9;
assign sum8 = a + b;
assign sum9 = a + b;
```

在第一个赋值语句中，所有操作数均为8位宽，所以会执行8位的加操作。加法的进位会被丢弃。第二个赋值语句中，由于sum9的位宽为9，信号a和b会被扩展到9位，然后执行9位的加操作。sum9得到的是结果中的进位。如果要保留进位，我们还可以使用连接运算符进行运算：

```
assign{c_out,sum8} = a + b;
```

虽然转换规则简单易懂，但一些细微差别很有可能导致错误。例如，假设a，b，sum1，sum2为8位数据。下面的语句将得出不同的结果：

```
// 将0 移位到sum1 的最高位
assign suml = (a + b) >>1;
// 将a + b 的溢出位移位到sum2 的最高位
assign sum2 = (0 + a + b) >>1;
```

第一句中，所有操作数为8位，所以执行的是8位的加运算。当移位操作执行时，进位会被丢弃，0会被移到最高位。第二句中，0为整数，32位宽，所有a和b会被扩展到32位后进行加法操作，然后进行加法和移位。结果会被截取为8位后赋值给sum2，因此sum2[7]存放的是进位。这种转换经常应用在有符号数据类型操作中（详见本书7.3节）。

可通过一种安全但有些繁琐的方式人工调整操作数的位宽。例如，获取sum2的另一种方式：

```
wire [8:0] sum_ext;// 将sum 扩展到9 位
...
assign sum_ext = {1'b0,a} + {1'bO,b};
assign sum2 = sum_ext[9:1];
```

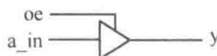
代码较长但不容易出错。

总之，我们必须注意 Verilog 的位长度自动调整机制。没意识到的位宽不匹配可能会导致细微的、不易发现的错误。除非一些无影响的调整，如整数 0 被置为全 0，我们应手动调整位长度，或者在文档中完整注释所期望的自动调整。

3.2.9 z、x 的综合

除了常规的逻辑 0 和逻辑 1，线网和变量还可以包含 z 和 x 这两种值。尽管不是运算符，我们仍安排在本节讨论这两种值的综合。

z 的综合 z 表示高阻或断开的电路结构。z 非通常的逻辑值，只能被综合为三态缓冲器。三态缓冲器的标识和功能如图 3-1 所示。缓冲器的操作由使能信号 oe（输出使能）控制，该信号为 1 时，输入端口传输到输出端口，反之，该信号为 0 时，输出端口 y 可看作是断开电路。三态缓冲器代码：



oe	y
0	Z
1	a_in

图 3-1 三态缓冲器的标识及功能表

```
assign y = (oe)? a_in:1'bz;
```

三态缓冲器最常见的应用是双向端口，用以实现对物理 I/O 引脚更为有效的利用。一个简单的例子如图 3-2 所示。信号 dir 控制 bi 引脚的信号流向。当 dir 为 0 时，三态缓冲器为高阻态，输出信号被阻断。该端口被用作输入端口，输入信号连至 sig_in 信号。当 dir 为 1 时，引脚用作输出端口，sig_out 信号连至外部电路。HDL 代码如下：

```
module bi_demo (
inout wire bi,
...
)
assign sig_out = output_expression;
...
```

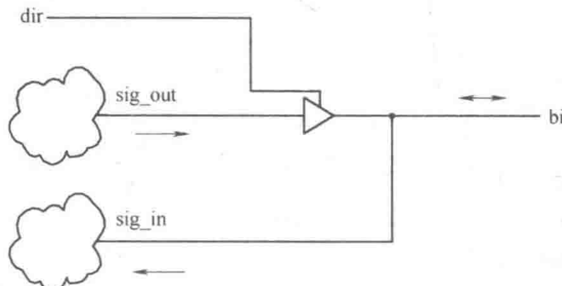


图 3-2 边界 I/O 端口信号缓冲器

```
assign some_signal = expression_with_sig_in;
...
assign bi = (dir)? sig_out:1'bz;
assign sig_in = bi;
...
```

注意，bi 端口类型一定要声明为 inout，用于双向操作，对于 Xilinx Spartan-3 器件，三态缓冲器只存在于物理引脚的 IOB 中。因此，三态缓冲器只能用于 I/O 端口的设计，也只能映射到 FPGA 器件的物理引脚上。

x 的综合在一些组合逻辑电路中，某些输入形式是不可能发生的，所以其输出值是无关紧要的。这时，我们会经常将输出赋值为“忽略”值。在综合时，为了更好地进行优化，“忽略”值会被赋为定值（0 或 1）。思考表 3-5 所示的真值表。假设 i 不可能为 11，因此其对应的输出被设定为“忽略”值。综合时，我们可以使用 x 代表忽略值。上表的一种代码实现为

```
assign y    = (i ==2'b00)? 1'b0:
              (i ==2'b01)? 1'b1:
              (i ==2'b10)? 1'b1:
              1'bx;//i ==2 'b11
```

表 3-5　真　值　表

输入 i	输出 y
0 0	0
0 1	1
1 0	1
1 1	x

虽然该方法可以帮助简化电路结构，但是会引入仿真和综合的不一致。仿真中，x 是 0 和 1 之外的独立值。如果仿真时输入为 11，输出会变为 x，这与综合后的结果（0 或 1）是不一致的。然而，由于原始规范要求任何情况下不应出现 11，因此 testbench 中出现的 x 可作为潜在错误的提示。

3.3　组合逻辑电路 always 块

为便于系统建模，Verilog 包含大量顺序执行的过程语句。由于这些语句的行为不同于通常的并行电路模型，因此要封装在 always 块或 initial 块中使用。initial 块只在仿真开始时执行一次。initial 可在仿真过程中使用，如示例 1.7 中所示。只有 always 块可以被综合，本节会进行详细的讨论。由于顺序语句更抽象，因此这种类型的代码常被称作行为描述。

always 块可以被当做黑盒子，其行为由内部的顺序语句描述。顺序语句包含很多结构，但许多是没有明确的硬件相对应的。不好的 always 块经常会导致不必要的复杂实现，或者根本不能被综合。本节将重点讨论组合逻辑电路的综合，我们仅限于以下 3 种语句的讨论：

- 阻塞顺序赋值；

- if 语句；
- case 语句。

后两种可以认为是推断布线结构的构建语句。

3.3.1 基本语法和行为

带有敏感列表（又称事件控制表达式）的 always 块简化语法：

```
always @ ([sensitivity-list])
    begin [optional name]
        [optional local variable declaration];
        [procedural statement];
        [procedural statement];
    ...
end
```

[sensitivity-list]是信号和事件的列表，always 块需要对这些信号和事件做出响应。对于组合逻辑电路，所有的输入信号都应该包含在敏感列表中。always 块可包含任意数量的顺序语句。如果只有一条语句，则 begin 和 end 可以省略。@([sensitivity-list])语句实际上是一种时序控制结构。在可综合的 always 块中，它通常是唯一的时序控制结构。

always 块可看作是一种复杂的电路组成部分，可被挂起或激活。当敏感列表中的任一信号发生变化或事件发生时，该部分电路会被激活并执行内部的顺序语句。由于没有其他的时序控制结构，执行过程会一直进行，直到最后一条语句完成，电路挂起。因此，always 块实际上是个“无限循环”，每个循环的开始由敏感列表控制。

3.3.2 顺序赋值语句

顺序赋值只能在 always 或 initial 块中使用，赋值语句有两种：阻塞赋值和非阻塞赋值。基本语法如下：

```
[variable-name] = [expression];// 阻塞赋值
[variable-name] <= [expression];// 非阻塞赋值
```

阻塞赋值，表达式的值经过计算后会在执行下条语句前立即赋值给变量（赋值过程“阻塞”了其他语句的执行）。其表现类似 C 语言中普通变量赋值。在非阻塞赋值中，表达式的值会在 always 块的最后进行赋值（赋值行为不会阻塞其他语句的执行）。

阻塞和非阻塞赋值经常令 Verilog 新手困惑，如果不理解这两种同赋值的区别，可能会导致无法预料的电路行为或者产生竞争冒险。使用的基本使用原则是：

- 对组合逻辑使用阻塞赋值；
- 对时序逻辑使用非阻塞赋值。

我们会在本书 7.1 节中详细讨论阻塞赋值和非阻塞赋值。由于本章的核心是组合逻辑电路，所以只会用到阻塞赋值语句。

3.3.3　变量数据类型

过程赋值的表达式只能赋值给变量，类型可以是 reg、integer、real、time 和 realtime。reg 数据类型和 wire 数据类型相似，但只能与过程赋值一起使用。Ingteger 数据类型表示固定宽度（通常是 32 位）的二进制补码格式的有符号数。由于位宽固定，通常不用其进行综合。其他的数据类型用于建模和仿真，不能被综合。

3.3.4　简单示例

我们用两个简单示例说明 always 块和阻塞赋值的用法。

1 位比较器　我们可以将示例 1.1 中的 1 位比较器电路重新用 always 块实现，如示例 3.1 所示。

示例 3.1　1 位比较器的 always 块实现

```
module eq1_always
  (
   input wire i0, i1,
   output reg eq       //eq 声明为reg 数据类型
  );
  //p0 和p1 声明为reg 数据类型
  reg p0, p1;
  always @ (i0, i1) //i0 和i1 应该在敏感列表中
  begin
    // 语句顺序很重要
    p0 = ~i0 & ~i1;
    p1 = i0 & i1;
    eq = p0|p1;
  end
endmodule
```

由于 eq、p0 和 p1 信号在 always 块中赋值，它们被声明为 reg 数据类型。敏感列表中包括 i0 和 i1，逗号分隔。当其中之一变化时，always 块被激活。3 个阻塞赋值语句会顺序执行，类似 C 程序语句。语句顺序很重要，且 p0 和 p1 在使用前应被赋值。

在 Verilog-1995 中，敏感列表中的逗号由关键字 or 替代。例如

always @ (a, b, c)

被写作

always @ (a or b or c)

本书中我们通常使用逗号。

组合逻辑电路必须将有输入信号包含在敏感列表内，以实现对期望的行为进行正确建模。缺失一个信号可能会导致综合和仿真的不一致。在 Verilog - 2001 中，我们可以使用符号：

always @ *

隐含包含所有的输入信号。本书中，我们会在组合逻辑电路中使用这种结构。

三输入与门示例 1. 1 和示例 3. 1 的相似会令人费解，连续赋值与过程语句是完全不同的。思考示例3. 2 中的代码，这是一段实现对 a、b 和 c 相与操作（即 a & b & c）的电路。

示例 3. 2　行为精减且使用一个变量的与操作电路

```
module and_block_assign
  (
   input wire a, b, c,
   output reg y
  );
  always @ *
      begin
    y = a;
    y = y & b;
    y = y & c;
  end
endmodule
```

综合后电路如图 3-3a 所示，如果我们用类似的方式连续赋值，如示例 3. 3 所示，描述是错误的。

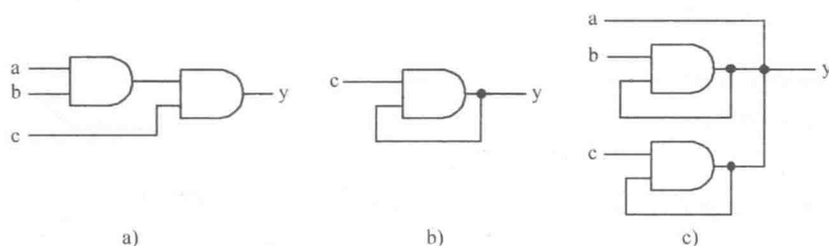


图 3-3　正确及错误的语句综合后电路

示例 3.3　精简的与门电路错误代码

```
module and_cont_assign
  (
   input wire a, b, c,
   output wire y
  );
  assign y = a;
  assign y = y & b;
  assign y = y & c;
endmodule
```

在以上代码中，每个连续赋值会综合成一个电路部分，y 在左边出现三次意味着三路输出被绑在一起。对应的电路如图 3-3c 所示，显然这不是我们想要的电路结构。

3.4 if 语句

3.4.1 语法

if 语句的简化语法为

```
if [bool-expr]
begin
    [procedural statement];
    [procedural statement];
end
else
```

```
begin
    [procedural statement];
    [procedural statement];
end
```

【boolean-expr】是布尔型表达式，会被最先计算。如果结果为真，会执行接下来的语句，否则，else 分支中的语句将被执行。else 分支是可选的，可以省略。如果分支中只有一条 procedural 语句，begin 和 end 也可以省略。

可将多个 if 语句进行“级联”，以实现多种条件及优先级的计算，如

```
If [Boolean_expr_1]
…
else if [Boolean_expr_2]
…
else if [Boolean_expr_3]
…
else
…
```

综合时，if 语句会推断出“优先级布线”网络，相关话题会在本书 3.6 节详细讨论。

3.4.2 示例

我们用两个简单的示例来说明 if 语句的用法。第一个例子是优先级编码器。优先级编码器有 4 个请求：r[4]、r[3]、r[2]和 r[1]，分组为一个 4 位的输入信号 r，r[4]拥有最高优先级。其功能表见表 3-6，HDL 代码如示例 3.4 所示。

表 3-6　四请求优先编码器功能表

输入 r				输出 pcode		
1	—	—	—	1	0	0
0	1	—	—	0	1	1
0	0	1	—	0	1	0
0	0	0	1	0	0	1
0	0	0	0	0	0	0

示例 3.4　使用 if 语句的优先编码器

```
module prio_encoder_if
  (
   input wire [4:1] r,
   output reg [2:0] y
  );
  always @ *
    if (r[4] ==1'b1)        //也可写为r[4]
```

```
  y = 3'b100;
else if (r[3] == 1'b1)       // 也可写为r[3]
  y = 3'b011;
else if (r[2] == 1'b1)       // 也可写为r[2]
  y = 3'b010;
else if (r[1] == 1'b1)       // 也可写为r[1]
  y = 3'b001;
else
  y = 3'b000;
endmodule
```

代码会首先检查 r[4]的请求，若请求有效则赋值 100 给输出。如果 r[4]的请求无效，就会接着检查 r[3]的请求，直到所有请求被检查。注意，当 r[4]为 1 时，布尔表达式(r[4] == l'bl)为真。由于在 Veriolg 中真值也会被表示为 1'bl，因此布尔表达式也可写作(r[4])。

表 3-7　带使能的 2:4 解码器真值表

输入			输出
性能	a(1)	a(0)	y
0	—	—	0 0 0 0
1	0	0	0 0 0 1
1	0	1	0 0 1 0
1	1	0	0 1 0 0
1	1	1	1 0 0 0

第二个例子是个二进制解码器。一个 $n:2^n$ 解码器根据输入组合置 2^n 位中的 1 位有效。表 3-7 给出了 2:4 解码器的真值表，电路还会带有一个控制信号 en，有效时使能该解码功能。HDL 代码详见示例 3.5。

示例 3.5　使用 if 语句表示的二进制解码器

```
module decoder_2_4_if
  (
   input wire [1:0] a,
   input wire en,
   output reg [3:0] y
  );
  always @ *
    if (en == 1'b0)          // 也可写为(~en)
      y = 4'b0000;
    else if (a == 2'b00)
      y = 4'b0001;
```

```
    else if (a ==2'b01)
      y =4'b0010;
    else if (a ==2'b10)
      y =4'b0100;
    else
      y =4'b1000;
endmodule
```

代码首先确认 en 是否有效，如果条件为假（即 en 为 1），程序会顺序测试 4 个二进制组合。注意布尔表达式（en ==1' b0）也可写作（ ~en）。

3.5 case 语句

3.5.1 语法

case 语句的简化语法如下：

```
case [case-expr]
    [item]:
    begin
        [procedural statement];
        [procedural statement];
        ...
    end
    [item]:
    begin
        [procedural statement];
        [procedural statement];
        ...
    end
    [item]:
    begin
        [procedural statement];
        [procedural statement];
        ...
    end
```

```
        ...
        default:
        begin
            [procedural statement];
            [procedural statement];
            ...
        end
    endcase
```

case 语句是一种多路选择语句，这种语句会将一系列[item]表达式与[case_expr]表达式进行比较。当某条分支的[item]值与[case_expr]匹配时，该分支的语句将被执行。如果有多条分支匹配时，将会执行第一条被匹配的分支语句。最后的 item 可以是可选的 default 关键字，它将覆盖所有[case_expr]表达式未指定的值。如果分支中只有一条顺序语句时，则 begin 和 and 分隔符可以省略。

3.5.2 示例

我们用相同的编码和解码器例子说明 case 语句的用法。表 3-7 为 2:4 解码器的真值表。示例 3.6 给出了用 case 语句实现的 HDL 代码。

示例 3.6 使用 case 语句的二进制解码器

```
module decoder_2_4_case
   (
   input wire [1:0] a,
   input wire en,
   output reg [3:0] y
   );
   always @ *
   case({en,a})
   3'b000, 3'b001, 3'b010, 3'b011: y =4'b0000;
   3'b100: y =4'b0001;
   3'b101: y =4'b0010;
   3'b110: y =4'b0100;
   3'b111: y =4'b1000; // 也可使用default 分支
   endcase
endmodule
```

如果所有值使用同样的表达式，我们可以将多个数据拼成一个表达式，如第 10 行。注意，{en,a}表达式的所有可能值都被覆盖。

优先级编码器的功能表详见表 3-6。HDL 代码如示例 3.7 所示。

示例 3.7　使用 case 语句的优先编码器

```
module prio_encoder_case
    (
    input wire [4:1] r,
    output reg [2:0] y
    );
    always @ *
    case(r)
    4'b1000, 4'b1001, 4'b1010, 4'b1011,
    4'b1100, 4'b1101, 4'b1110, 4'b1111:
    y =3'b100;
    4'b0100, 4'b0101, 4'b0110, 4'b0111:
    y =3'b011;
    4'b0010, 4'b0011:
    y =3'b010;
    4'b0001:
    y =3'b001;
    4'b0000:  // 也可使用default 分支
    y =3'b000;
    endcase
endmodule
```

3.5.3　casez 和 casex 语句

除了常规的 case 语句外，还有其他两种 case 表达式。在 casez 语句中，z 值和分支表达式中的? 会被忽略（即，对应位无需匹配）。在 casex 语句中，z、x 及分支表达式中? 会被忽略。因为 z 和 x 在仿真中可能出现，? 符号会经常被使用。例如，上面的优先编码器代码可以使用 casez 语句编写，如示例 3.8 所示。

示例 3.8　使用 casez 语句的优先编码器

```
module
```

```
  (
    input wire [4: 1] r,
    output reg [2: 0] y
    );
    always @ *
    casez (r)
    4' b1???: y =3' b100;
    4' b01??: y =3' b011;
    4' b001?: y =3' b010;
    4' b0001: y =3' b001;
    4' b0000: y =3' b000;    // 也可使用default 分支
    endcase
endmodule
```

3.5.4 full case 与 parallel case

在 Verilog 中，[item]表达式不需要包括[case-expr]表达式的所有值，并且某些值还可能会被匹配多次。如下面的 casez 语句：

```
reg[2:0] s
...
casez (s)
    3'b111: y =1'b1;
    3'b 1??: y  =1'b0;
    3'b000: y =1'b1;
endcase
```

在上面的语句中，3'b111 会被匹配两次(一次在 3'b111 中，一次在 3'b1?? 中)。由于第一次匹配有效，如果 s 为 3'b111，y 结果为 1，当 s 为 3'b001、3'b010 或者 3'b011 时，无匹配项，y 将会“保持原值”。

当[case-expr]表达式的所有可能值都被[item]分支表达式覆盖，这种 case 语句称之为 full case；对组合逻辑电路来说，每一种输入组合都应有一个输出值相对应，因此必须使用 full case。可以通过增加 default 分支覆盖所有的未匹配值。例如，以上语句可以修改为

```
casez (s)
3'b111: y =1'b1;
3'b1??: y =1'b0;
```

```
        default: y = 1'b1; // 对于未指定分支表达式值,y 为1
    endcase
    case z (s)
        3'b111: y = 1'b1;
        3'b1??: y = 1'b0;
        3'b000: y = 1'b1;
        default: y = 1'bx; //y 为可忽略值
    endcase
```

当分支表达式中的值都互斥（即某值只能出现在一个分支中）时，称之为 parallel case。例如，上面的 case 语句不是 parallel case，因为 3'b111 出现在两个分支中。示例 3.6 和示例 3.7 中的 case 语句为 parallel case。

综合时，parallel case 语句通常推断为多路开关布线网络，非 parallel case 语句通常推断为优先级布线网络。下节我们会详细介绍。

许多综合工具都有 "full case 指令" 和 "parallel case 指令"，使用这些指令时，所有的 case 语句会被当做 full case 或 parallel case 进行综合。Verilog-2001 也具有相似的特性来实现该目的。使用这类指令会改变 Verilog 代码本来的语义，导致仿真和综合的不一致。本书中，我们使用代码实现 "full case" 和 "parallel case"，而非通过指令或属性来实现。

3.6 条件控制语句的布线结构

我们介绍了几种条件控制语句，包括?: 运算符、if 和 case 语句。在 C 语言中，这些结构都是顺序执行。在组合逻辑电路中是没有这种 "顺序" 控制的。这些结构通过路由网络实现。所有表达式同时计算，路由网络把结果路由到输出端。布线结构包括两种：优先路由网络和多路选择网络。这两种结构通常分别由 if-else 语句和 parallel case 语句生成。

3.6.1 优先路由网络

优先路由网络由一连串的 2 选 1 多路选择器实现。2 选 1 多路选择器的真值表和电路图如图 3-4a 所示。if-else 语句会被推断为优先路由网络。如下面的语句：

```
if (m == n)
    r = a + b + c;
    else if (m > n)
    r = a - b;
```

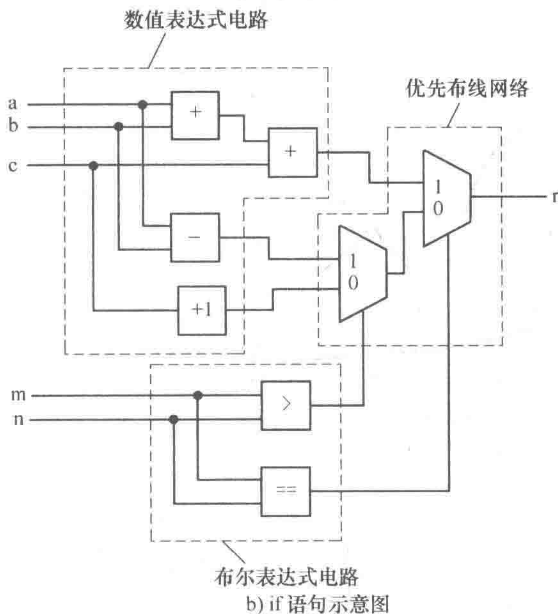
```
else
    r = c + 1;
```

电路框图如图 3-4b 所示。两个 2:1 多路选择器形成了优先路由网络，其他部分实现了各种布尔及算术表达式。如果第一个布尔条件表达式（即 m == n）为真，r 输出为 a + b + c，否则，端口 0 的数据会输出到 r。第二个布尔条件表达式（即 m > n）决定了 a - b 还是 c + 1 路由到输出。



sel	y
0(false)	i0
1(true)	i1

a) 2:1多路选择器示意图



b) if 语句示意图

图 3-4　if 语句的实现

注意，所有的布尔表达式和算术表达式的计算是并行的，布尔表达式的输出作为多路选择器的选择信号，用于将期望结果路由至 r。级联级数与 if-else 语句的个数成正比。大量使用 if-else 语句会导致很长的级联链路并导致较大的传播延时。

条件运算符（?:）类似一种简化的 if-else 语句并能够生成相似的优先路由网络。非 parallel case 语句会对第一个匹配分支设置一个优先权，因此也会生成相似的优先路由网络。例如下面的 case 语句：

```
case( expr )
    item1 : statement1;
```

```
    item2  : statement2;
    item3  : statement3;
    default: statement4;
endcase
```

它可以被转化为

```
    if[ expr == item1 ]
statement1;
elseif[ expr == item2 ]
statement2;
elseif[ expr == item3 ]
statement3;
else
statement4;
```

3.6.2　多路选择网络

多路选择网络是通过 *n* 选 1 多路选择器实现的。期望的输入端口被选择信号选定，这样相应的输入端便被路由到输出端。2^2 选 1 多路选择器示意图和真值表如图 3-5a 所示。

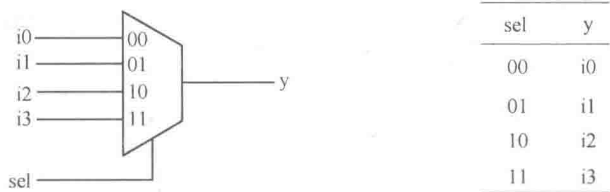
在 parallel case 语句中，我们可以把 case 表达式的每个值都映射为多路选择器的一个输入端口，并将相应结果连接到端口上。case 表达式被选择信号连接。可以通过一个例子解释。如下面 case 语句：

```
wire [1:0] sel;
...
case(sel)
    2'b00: r = a + b + c;
    2'b10: r = a - b;
    default: r = c + 1; //2'b01, 2'bll
endcase
```

该语句示意图如图 3-5b 所示。假设 sel 变量有 4 种可能的值：00、01、10 和 11。意味着一个 sel 作为选择信号的 2^2 选 1 多路选择器。当 sel 为 00 时，a + b + c 所得的结果被路由到 r 端，当 sel 为 10 时，a - b 所得的结果被路由到 r 端，当 sel 为 10 或者 11 时，c + 1 所得的结果被路由到 r 端。

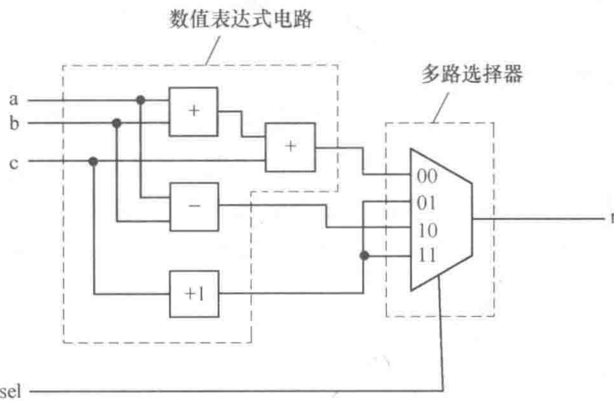
再次注意，所有数值表达式的计算都是并行的。Sel 变量被用作选择信号将期望结果路由至输出端。多路选择器宽度（即输入端口数量）随着 sel 信号呈几何增长。

总的来说,当优先权在特定的条件下给定时,带有优先级的路由网络更为有效,如优先编码器,对于真值表或功能表,多路选择网络更为有效,如二进制解码器。



sel	y
00	i0
01	i1
10	i2
11	i3

a) 4选1多路选择器示意图和功能表



b) parallel case语句示意图

图 3-5　parallel case 语句实现

3.7　always 块的通用编码准则

Verilog 可用于建模和综合。在编写用于综合的代码时，我们要注意不同的语言结构是如何映射到硬件的，这一点对于 always 块尤为重要，因为变量和顺序语句可以在块内使用。我们要牢记：代码的目标是生成硬件电路，而不是用 C 描述一个顺序算法。不这样做会经常导致一些不可综合的代码、不必要的复杂实现或仿真和综合之间的差异。在本节中，我们回顾一些常见的错误，并推荐一些编码准则。

3.7.1　组合逻辑电路代码的常见错误

以下是在组合逻辑电路代码中常见的错误：

- 同一变量被多个 always 块赋值；
- 敏感列表不完整；
- 不完全的分支和不完整的输出赋值。

这些错误会在后续内容中讨论。

同一变量被多个 always 块赋值 在 Verilog 中，变量是允许被多个 always 块赋值（即出现在表达式左侧）的。例如，下面的代码片段中，y 变量在两个 always 块中被赋值：

```
reg y;
reg a,b,clear;
...
always @ *
    if(clear)y = 1'b0;
always @ *
    y = a&b;
```

尽管代码语法正确，能被仿真，但却不能被综合。回想一下，每个 always 块均可被翻译为一个电路部件。上面的代码表明，y 是两个电路部件的输出，且能被每个部件更新。没有任何物理电路能够表现这种行为，因此代码是不可综合的。我们必须把赋值语句分组到一个单独的 always 块中，如

```
always @ *
    if(clear)
    y = 1'b0;
else
    y = a&b;
```

敏感列表不完整 对于一个组合逻辑电路而言，输出是输入的函数，因此输入信号的任何变化都应激活电路。这意味着，所有的输入信号都应包括在敏感列表中。例如，一个两输入的与门可写为

```
always @ (a, b) //a 和b 都在敏感列表中
    y = a&b;
```

如果我们忘记包括 b，代码会变成

```
always @ (a) //b 从敏感列表中丢失
    y = a&b;
```

虽然代码在语法上仍然正确，但是其行为却完全不同。当 a 发生变化，always 块被激活，y 得到 a&b 的值。当 b 改变，always 块依然被挂起，因为它对 b 是不“敏感”的，y 保持之前的值。没有物理电路能体现这种行为。作为替代，大多数综合软件将发出警告信息，并推断出与门。然而仿真软件仍会模拟预期的行为，因此导致了仿真和综合之间的差异。

Verilog-2001 引入了一个特殊的符号，@ *，用于隐式包括所有相关的输入信号，从而消除了这个问题。对组合逻辑电路描述使用这个符号是一个很好的做法。

不完全分支和不完整的输出赋值组合逻辑电路的输出是个只与电路输入有关

的函数，不应该包含任何内部状态（即存储器）。always 块中一个常见的错误就是在组合逻辑电路中生成存储器。Verilog 标准规定，在 always 块中如果变量没有被赋值，它将保持原值。在综合过程中，将会形成内部状态（一个闭合反馈环）或存储元件（如一个锁存器）。

为了防止 always 块中无意的存储器的产生，所有的输出信号在任何时间都要被赋予一个合理值。不完全分支和不完整的输出赋值是导致产生存储器的两个常见错误，为了避免这些，我们在开发组合逻辑电路的代码时应该遵守以下规则：

- if 或 case 语句应该包含所有的分支；
- 在每一个分支为每一个输出信号赋值。

思考下面的这段代码，它描述的是一个产生大于有效（即 GT）输出信号和等于有效（即 EQ）输出信号的电路。

```
always @ *
    if(a > b)// 在这个分支中eq 未被赋值
        gt = 1'b1;
    else if(a == b) // 在这个分支中gt 未被赋值
        eq = 1'b1;
    // 最后else 分支被遗漏
```

该段代码违反了这两条规则。

让我们先查看不完全分支的错误。代码片段中没有 else 分支。如果两个 a > b 和 a == b 的表达式都是假的，gt 和 eq 都没有被赋值。根据 Verilog 的定义，它们将保持以前的值（即输出取决于内部状态）并生成锁存器。

代码片段还有不完整输出赋值的错误。例如，当 a > b 的表达式是真实的，eq 没有被赋值，这样便会保持以前的状态，从而生成锁存器。

有两种方法可以修改此错误。第一种是添加 else 分支，并为所有的输出变量显式赋值。代码变为

```
always @ *
    if (a > b)
        begin
            gt = 1'b1;
            eq = 1'b0;
        end
    else if (a == b)
        begin
            gt = 1'b0;
            eq = 1'b1;
```

```
        end
    else// 即a < b
        begin
            gt = 1'b0;
            eq = 1'b0;
        end
```

另一种方法是在 always 块的开始为每一个变量指定一个默认值，用来覆盖未指定的分支和未赋值的变量。代码会变成：

```
always @ *
    begin
        gt = 1'b0; //gt 默认值
        eq = 1'b0;//eq 默认值
    if (a > b)
        gt = 1'b1;
    else if (a == b)
        eq = 1'b1;
    end
```

如果 gt 和 eq 在后面没有被赋值，那么它们都会被赋值为 0。

如果某些[case-expr]所表示的值没有被分支表达式覆盖到，case 语句会遇到相同的错误（也就是说，不是一个完整的 case 语句）。思考下面的代码片段：

```
reg [1:0] s
...
case (s)
    2'b00: y = 1'b1;
    2'b10: y = 1'b0;
    2'b11: y = 1'b1;
endcase
```

值 2'b01 不包括在任何一个分支中。如果 s 出现这个组合，y 将保持其先前的值，并产生一个意料之外的锁存器。要修正这个错误必须确保 y 在任何时刻都被赋值。方法之一是在末尾使用 default 的关键字来覆盖所有未指定的值。我们还可以替换最后一个分支表达式：

```
case (s)
    2'b00: y = 1'b1;
    2'b10: y = 1'b0;
    default: y = 1'b1; // 在2'b01 状态时y 的值为1
```

```
endcase
```

或将一个忽略值添加到新的分支中：

```
case (s)
    2'b00: y = 1'b1;
    2'b10: y = 1'b0;
    2'b11: y = 1'b1;
    default: y = 1'bx; // 在2 'b01 状态时y 的值为不定态x
endcase
```

或者在 always 块的最开始指定一个默认值：

```
    y = 1'0; // 也可使用y = 1 'bx 用于产生忽略值
    case (s)
        2'b00: y = 1'b1;
        2'b10: y = 1'b0;
        2'b11: y = 1'b1;
    endcase
```

3.7.2　准则

always 块是一个灵活而强大的语言结构。但是，须谨慎使用，在生成正确有效电路的同时，还需避免综合和仿真之间的差异。以下是组合逻辑电路的编码准则：

- 同一变量的赋值只能发生在单一的 always 块中；
- 组合逻辑电路中使用阻塞赋值；
- 使用@ * 自动将所有输入包括在敏感列表中；
- 确保 if 和 case 语句涵盖了所有分支；
- 确保在所有分支中，输出都被赋值；
- 满足之前的两个准则的方法之一是在 always 块的开始为所有输出指定默认值；
- 用代码描述 full case 和 parallel case 语句，而不是用软件的指令或属性；
- 注意由不同的控制结构产生的路由网络类型；
- 思考硬件结构，而不是 C 代码。

3.8　参数和常量

3.8.1　常量

HDL 代码中经常在表达式和数组边界上使用常数值。这些值在模块内是固

定的，不能被修改。一个好的设计实践是：用符号常量取代“固定值”。这会使代码清晰并有助于将来的维护和修改。在 Verilog 中，常量可以使用 localparam（“本地参数”）关键字来声明。例如，我们可以声明一个数据总线的宽度和范围：

```
        localparam DATA_WIDTH =8,
                   DATA_RANGE =2 * * DATA_WIDTH -1;
```

或者定义一个符号端口的名称：

```
    localparam UART_PORT   =4'b0001,
               LCD_PORT    =4'b0010,
               MOUSE_PORT  =4'b0100;
```

声明中的表达式，如 2 ** DATA-WIDTH-1，会在预处理时被计算，因此并没有推断物理电路。在本书中，我们使用大写字母表示常量。

常量的用法最好通过一个例子来说明。思考下面进位加法器的代码，一种实现手段是将输入位宽手动扩展 1 位，执行常规的加法，并将和的最高位作为进位。代码如示例 3.9 所示。

示例 3.9　使用固定名字的加法器

```
module adder_carry_hard_lit
      (
    input wire [3: 0] a, b,
    output wire [3: 0] sum,
    output wire cout // 进位
    );
    // 信号声明
    wire [4: 0] sum_ext;
    // 实体
    assign sum_ext = {1' b0, a} + {1' b0, b};
    assign sum = sum_ext [3: 0];
    assign cout = sum_ext [4];
endmodule
```

这是一个 4 位加法器的代码。固定位宽，例如用 3 和 4 表示位宽范围，如 wire [4: 0]、sum_ext [3: 0] 和最高位 sum_ext [4]。如果我们想把它修改成一个 8 位加法器的代码，不得不手动修改这些名字。如果代码很复杂或名字用在许多地方，这将是一个繁琐且容易出错的过程。

为了提高可读性，我们可以使用符号常数 N 来表示加法器的位数。修改后的代码如示例 3.10 所示。

示例 3.10 使用常量的加法器

```
module adder_carry_local_par
    (
    input wire [3:0] a, b,
    output wire [3:0] sum,
    output wire cout // 进位
    );
    // 常量声明
    localparam N = 4,
    N1 = N - 1;
    // 信号声明
    wire [N:0] sum_ext;
    // 实体
    assign sum_ext = {1'b0, a} + {1'b0, b};
    assign sum = sum_ext [N1:0];
    assign cout = sum_ext [N];
endmodule
```

常量使代码更易于理解和维护。

3.8.2 参数

Verilog 模块可以被实例化为一个原件，成为更大设计的一部分，如第 1.6 节所讨论的。Verilog 提供了一种结构，称之为 parameter（参数），用于将信息传递到一个模块。这种机制使得该模块通用性和可重用性成为可能。参数不能被模块内部修改，因此它的功能就像一个常数。

在 Verilog-2001 中，parameter 的声明部分可以被添加在头部、引脚声明之前。其简化的语法如下：

```
module [module_name]
#(
  parameter [parameter_name] = [default_value],
     ...
     [parameter_name] = [default_value];
```

```
)
(
...//I/O port declaration
);
```

例如，前面的加法器代码可以被修改为使用参数作为加法器宽度，如示例 3.11 所示。

示例 3.11　使用参数的加法器

```
module adder_carry_para
  #(parameter N =4)
  (
    input wire [N-1:0] a, b,
    output wire [N-1:0] sum,
    output wire cout   // 进位
  );
  // 常量声明
  localparam N1 = N-1;
  // 信号定义
  wire [N:0] sum_ext;
  // 实体
  assign sum_ext = {1'b0, a} + {1'b0, b};
  assign sum = sum_ext[N1:0];
  assign cout = sum_ext[N];
endmodule
```

N 被声明成默认值为 4 的参数。N 被声明后，可以像常数一样应用于端口声明和模块主体。

如果加法器后续用于其他代码中的元件，可以在例化元件时将期望的值赋值给参数，并覆盖默认值。类似于第 1.6 节中讨论的端口连接，参数赋值可以通过名称或顺序列表的方式完成。第 1.6 节中讨论的顺序列表方式存在一个潜在问题，本书会坚持使用按名称赋值的方式。如果参数的赋值被省略，则将使用默认值。元件例化时参数的使用如示例 3.12 所示。

示例 3.12　加法器例化例子

```
module adder_insta
```

```
    (
    input wire [3: 0] a4, b4,
    output wire [3: 0] sum4,
    output wire c4,
    input wire [7: 0] a8, b8,
    output wire [7: 0] sum8,
    output wire c8
    );
    // 初始化8 位加法器
    Adder_carry_para# (.N (8)) unit1
        (.a (a8), .b (b8), .sum (sum8), .cout (c8)) ;
    // 初始化4 位加法器
    Adder_carry_para unit2
        (.a (a4), .b (b4), .sum (sum4), .cout (c4));
endmodule
```

参数提供了一种创建可扩展代码的机制，电路的“位宽”可以调整，以满足特定的需求。这使得代码更具可移植性，并促进了设计重用。

3.8.3 Verilog-1995 的参数使用

前面讨论的 localparam 关键字，在开头声明，且通过名称赋值是 Verilog-2001 中一个新特性。在 Verilog-1995 中，参数在开头之后声明，并且只能使顺序清单方式或 defparam 语句重新定义。另外，常量必须声明为参数，即使它们不应被重新定义。Verilog-1995 语法下的加法器代码实现如示例 3.13 所示。

示例 3.13　Verilog-1995 参数的使用

```
module adder_carry_95 (a, b, sum, cout);
    parameter N =4; // 在端口前声明参数
    parameter N1 = N - 1; // 在 Verilog_1995 中无localparam 声明
    input wire [N1:0] a, b ;
    output wire [N1:0] sum;
    output wire cout;
    // 信号定义
    wire [N:0] sum_ext;
    // 实体
```

```
    assign sum_ext = {1'b0, a} + {1'b0,b};
    assign sum = sum_ext [N1:0];
    assign cout = sum_ext [N];
endmodule
```

当一个模块被实例化时，参数只能通过顺序列表的方式重新定义，如

```
        adder_carry_95 #(8,7) unit1
        (.a(a8),.b(b8),.sum(sum8),.cout (c8))
```

或者通过 defparam 方式，如

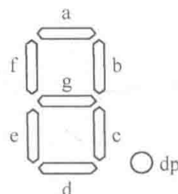
```
    defparam unit1.N =8;
    defparam unit1.N1 =7;
    adder-carry-95 unit1
        (.a(a8),.b(b8),.sum(sum8),.cout(c8));
```

Verilog-1995 的方案比较繁琐，并且可能引入细微误差，在这本书中我们不使用它。

3.9 设计实例

3.9.1 7 段 LED 数码管十六进制译码器

7 段 LED 显示简图如图 3-6a 所示。它由 7 个 LED 条和一个圆形的 LED 小数点组成。在开发板上，7 段 LED 配置为低电平有效，即控制信号为 0 时，LED 点亮。



a) 七段LED显示原理图



b) 十六进制数字样式

图 3-6 七段 LED 数码管显示和十六进制样式

7 段 LED 数码管十六进制译码器将 4bit 输入作为十六进制数字，译码成相应的 LED 样式，如图 3-6b 所示。出于完整性考虑，我们假设有一个 1bit 输入信号 dp，直接连接在小数点的 LED 上。LED 控制信号 dp、a、b、c、d、e、f 和 g，被组合为一个独立的 8bit sseg 信号。代码如示例 3.14 所示。使用一个 case 语句来列出所有七个 LSB 的 sseg 信号样式。MSB 连至 DP。

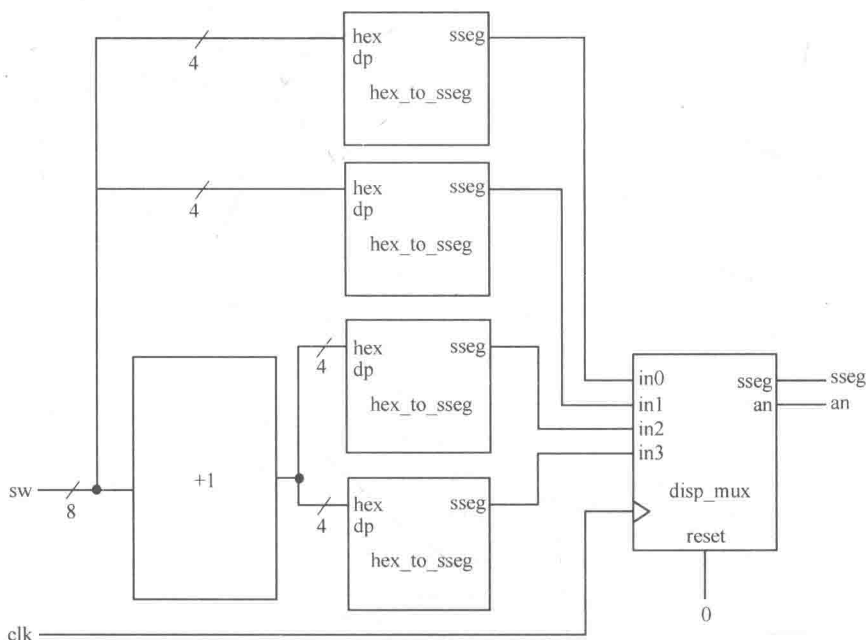
示例 3.14　七段十六进制 LED 数码管译码器

```
module hex_to_sseg
    (
    input wire [3:0] hex,
    input wire dp,
    output reg [7:0] sseg // 输出低有效
    );
    always @ *
    begin
        case (hex)
            4'h0:sseg [6: 0] =7'b0000001;
            4'h1:sseg [6:0] =7'b1001111;
            4'h2:sseg [6:0] =7'b0010010;
            4'h3:sseg [6:0] =7'b0000110;
            4'h4:sseg [6:0] =7'b1001100;
            4'h5:sseg [6:0] =7'b0100100;
            4'h6:sseg [6:01 =7'b0100000;
            4'h7:sseg [6:0] =7'b0001i11;
            4'h8:sseg [6:0] =7'b0000000;
            4'h9:sseg [6:0] =7'b0000100;
            4'ha:sseg [6:0] =7'b0001000;
            4'hb:sseg [6:0] =7'b1100000;
            4'hc:sseg [6:0] =7'b0110001;
            4'hd:sseg [6:0] =7'b1000010;
            4'he:sseg [6:0] =7'b0110000;
            default : sseg [6:0] =7'b0111000; //4 'hf
    endcase
    sseg[7] =dp;
endmodule
```

开发板上共有 4 个 7 段 LED 显示器，为了节省 FPGA 芯片的 I/O 引脚的数量，使用分时复用机制。分时复用模块 disp-mux 如图 3-7a 所示。输入是 in0，in1，in2 和 in3，分别对应 4 个 8 位七段 LED 的数据，输出是一个 4 位信号 an，用于使能四个单独的显示器，sseg 是共享的 8bit 信号，用于控制 8 个 LED 段。电路产生定时使能信号，将 4 个输入数据交替路由到输出端。该模块的设计将在第 4 章讨论。现在，我们只把它作为一个需要 4 个 7 段 LED 数据的黑盒子，并在代码中将其实例化。



a) LED分时选择模块框图



b) 译码器测试电路框图

图 3-7 LED 分时选择模块和译码器测试电路

测试电路 我们使用一个简单的 8 位累加电路来验证译码器的操作。如图 3-7b 所示。输入 sw 是开发板中的 8 位开关。它被反馈到一个累加器上，用于获取 sw + 1。原始和累加的 sw 信号被传递至 4 个译码器中，用于将 4 个十六进制数字显示在 7 段 LED 显示器上。代码如示例 3. 15 所示。

示例3.15　Hex-to-LED 解码器测试电路

```
module hex_to_sseg_test
    (
    input wire clk,
    input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
    );
    // 信号定义
    wire [7:0] inc;
    wire [7: 0] led0, led1, led2, led3;
        // 输入加1
        assign inc = sw + 1;
        // 例化四个十六进制解码器
        // 例化输入的4 个最低位
        Hex_to_sseg sseg_unit_0
        (. hex(sw[3:0]),. dp(1'b0),. sseg(led0));
        // 例化输入的4 个最高位
        Hex_to_sseg sseg_unit_1
        (. hex(sw[7:4]),. dp(1'b0),. sseg(ledl));
        // 例化增量值的4 个最低位
            Hex_to_sseg sseg_unit_2
        (. hex(inc[3:0]),. dp(1'bl),. sseg(led2));
        // 例化增量值的4 个最高位
        Hex_to_sseg sseg_unit_3
        (. hex(inc[7:4]),. dp(1'bl),. sseg(led3));
        // 例化7 段LED 时间码显示模块
        Disp_mux disp_unit
        (. clk(clk),. reset(1'b0),. in0(led0),. in1(led1),
        . in2(led2),. in3(led3),. an(an),. sseg(sseg));
    endmodule
```

可以将第2章中的程序进行综合并下载到开发板上。注意，在综合过程中，含有定时选择模块的代码 disp-mux. v 文件和 ucf 约束文件必须在 Xilinx ISE

工程中。

3.9.2 “符号—幅值”加法器

整数可以用“符号—幅值”格式表示，其中的最高位是符号，剩余位表示数值。例如，3 和 -3 可以用 4 位的符号—幅值格式表示为“0011”和“1011”。

“符号—幅值”加法器用于完成这种格式的加法操作，其实现方式概述如下：

- 如果两个操作数的符号相同，数值相加，符号保持不变；
- 如果两个操作数的符号不同，从较大的数值中减去较小的数值，并保持较大值的符号。

一种实现方式是将电路行为分成两个阶段。第一阶段是根据数值大小将两个输入数据分为 max 信号和 min 信号。第二阶段查看符号位，并对数值进行加法或者减法运算。需要注意的是，由于两个信号已经分为 max 信号和 min 信号，max 数值总是大于 min，最终的符号是 max 的符号。

代码如示例 3.16 所示，它采用了两个阶段的实现方案。为清晰起见，我们在内部将输入数据的符号和数值分开。参数 N 用来表示加法器的宽度。

示例 3.16 “符号—幅值”数加法器

```
module sign_mag_add
    #(
    parameter N = 4
    )
    (
        input wire [N-1:0] a, b,
        output reg [N-1:0] sum
    );
    // 信号定义
    reg [N-2:0] mag_a, mag_b, mag_sum, max, min;
    reg sign_a, sign_b, sign_sum;
    // 实体
    always @ *
    begin
        // 分离数据和符号位
        mag_a = a[N-2:0];
```

```
        mag_b = b [N - 2:0];
        sign_a = a[N - 1];
        sign_b = b [N - 1];
        // 根据数据排序
        if (mag_a > mag_b)
            begin
                max = mag_a;
                min = mag_b;
                sign_sum = sign_a;
            end
        else
            begin
                max = mag_b;
                min = mag_a;
                sign_sum = sign_b;
            end
        // 数据加/减
        if (sign_a == sign_b)
            mag_sum = max + min;
        else
            mag_sum = max - min;
        // 形成输出
        sum = (sign_sum, mag_sum);
    end
    endmodule
```

测试电路　我们使用一个4位的“符号—数值”加法器来验证电路是否正常工作。测试电路如图3-8所示。两个输入数字连接至8位开关上，符号和数值显示在两个七段LED数码管上。两个按钮作为选择信号的多路开关，将操作数或和数路由至显示电路。最右边的七段LED显示3bit的数值，前面添加了一个0，并送到十六进制七段LED译码器中。下一个LED显示的是符号位，空白时表示正，亮起中间LED段表示负。这两个LED显示值被发送到3.9.1节中所示的时分复用模块disp-mux中。代码如示例3.17所示。

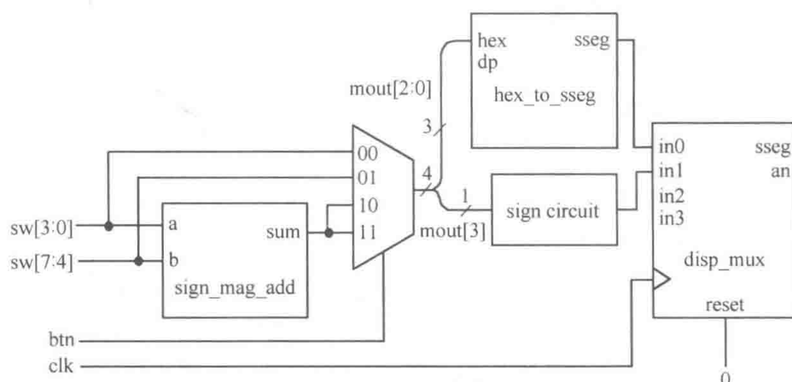


图 3-8　“符号—幅值”加法器测试电路

示例 3.17　“符号—幅值”加法器测试电路

```
module sm_add_test
   (
   input wire clk,
   input wire [1: 0] btn,
   input wire [7: 0] sw,
   output wire [3: 0] an,
   output wire [7: 0] sseg
   );
   // 信号定义
   wire [3: 0] sum, mout, oct;
   wire [7: 0] led3, led2, led1, led0;
   // 例化加法器
   Sign_mag_add # (.N (4)) sm_adder_unit
    (.a (sw [3: 0]), .b (sw [7: 4]), sum (sum));
   // 数值显示在最右边的7 段LED 上
   assign mout = (btn ==2' b00) ? sw [3: 0]:
    (btn ==2' b01) ? sw [7: 4]:
   sum;
   assign oct = {1' b0, mout [2: 0]};
   // 例化十六进制译码器
   Hex_to_sseg sseg_unit
    (.hex (oct), .dp (1' b0), .sseg (led0));
```

```
    // 在第二个7 段LED 上显示符号位
    // 中间LED 显示负数
    assign led1 = mout [3] ? 8' b11111110: 8' b11111111;
    // 其他2 个LED 空闲
    assign led2 = 8' b11111111;
    assign led3 = 8' b11111111;
    // 例化7 段时间码显示LED
    Disp_mux disp_unit
        (.clk (clk), .reset (1' b0), .in0 (led0), .in1 (led1),
    .in2 (led2), .in3 (led3), .an (an), .sseg (sseg));
endmodule
```

3.9.3 桶式移位器

虽然 Verilog 有内置的移位功能，但却没有轮转操作。在本节中，我们将检验一个可以让任意比特数的数据向右轮转的 8 位桶式移位器。该电路具有一个 8 位的数据输入端 a，和一个用来指定旋转量的 3 位控制信号 amt。第一个设计使用了 case 语句，详尽地罗列了 amt 信号的所有的组合及相应的轮转结果。代码如示例 3.18 所示。

示例 3.18　case 语句的桶式移位器

```
module barrel_shifter_case
    (
    input wire [7:0] a,
    input wire [2:0] amt,
    output reg [7:0] y
    );
    // 实体
    always @ *
    case (amt)
        3'00: y = a;
        3'01: y = {a[0], a[7:1]};
        3'02: y = {a[1:0], a[7:2]};
        3'03: y = {a[2:0], a[7:3]};
        3'04: y = {a[3:0], a[7:4]};
```

```
        3'05: y = {a[4:0], a[7:5]};
        3'06: y = {a[5:0], a[7:6]};
        default : y = {a[6:0], a[7]};
    endcase
endmodule
```

虽然代码很简单，但是随着数据位数的增加，代码会变得很冗长。另外，带有大量分支项的 case 语句意味着很宽的多路选择器，这会使得综合变的困难，并导致很大的传播延迟。作为选择，我们可以分级构造电路。在第 n 级，输入信号或被直接传递到输出，或者向轮转"2^n"个位置。第 n 级通过 amt 的信号的第 n 位控制。假设 3bit 的 amt 为 $m_2m_1m_0$。经过三级的轮转总量是 $m_22^2+m_12^1+m_02^0$，正是期望的轮转总量。本方案的代码如示例 3.19 所示。

示例 3.19　多级桶式移位器

```
module barrel_shif ter_stage
    (
    input wire [7:0] a,
    input wire [2:0] amt,
    output wire [7:0] y
    );
    // 信号定义
    wire [7:0] s0, s1;
    // 实体
    //stage 0 ,移0 位或1 位
    assign s0 = amt [0] ? {a[0], a[7:1]} :a;
    //stage 1 ,移0 位或2 位
    assign s1 = amt [1] ? {s0[1:0], s0[7:2]} :s0;
    //stage 2 ,移0 位或4 位
    assign y = amt [2] ? {s1[3:0], sl[7:4]} :s1;
endmodule
```

测试电路　要测试这个电路，我们可以使用 8 位的开关作为 a 信号，3 个按钮当作 amt 信号，8 个分立的 LED 灯用于输出。不用重新编写引脚分配约束文件，我们创建一个新的 HDL 文件将桶式移位器电路封装起来，并将它的信号映射到原型设计电路板的信号上。代码如示例 3.20 所示。

示例3.20　桶式移位器测试电路

```
module shif_ter_test
    (
    input wire [2:0]btn,
    input wire [7:0] sw,
    output wire [7:0] led
    );
    // 例化移位器
    barrel_shifter_stage shift_unit
    (.a(sw),.amt(btn),.y(led));
endmodule
```

3.9.4　简化的浮点加法器

浮点数是用来表示数据另一种格式。在相同的比特数的情况下，浮点数格式的表示范围远远大于有符号整型数格式。虽然 VHDL 具有内置的浮点数据类型，但是它实在太复杂，无法被自动综合。

对浮点数表示方法的详细讨论已经超出了本书的范围。本例使用了一个简化的13位格式，并忽略其舍入误差。这种表示方法包含一个符号位 s，表示数字的符号（1表示负数）；一个4位的指数域 e，它表示指数；一个8位的尾数字段 f，它代表有效位数或小数。在这种格式中，浮点数的值是 $(-1)^s * .f * 2^e$。$f * 2^e$ 是数字的幅值，$(-1)^s$ 只是一种表示符号的正规方式，说明“s 等于1意味着一个负数。”由于符号位是从数字的剩余部分分离出来的，浮点表示法可以看作为“符号-数值”格式的一种演变。

我们还可以做如下假设：

- 指数和尾数域都是无符号格式的；
- 数值的表示必须为正规方式或为零，正规方式表示意味着有效字段的 MSB 必须为1，如果计算结果的数值小于最小的非零数值 $0.10000000 * 2^{0000}$，就必须将它转换为零。

基于这些假设，最大和最小的非零幅值为 $0.11111111 * 2^{1111}$ 和 $0.10000000 * 2^{0000}$，范围约为 2^{16}（即 $(0.11111111 * 2^{1111})/(0.10000000 * 2^{0000})$）。

我们的浮点加法器设计遵循了用科学记数法手动相加数字的过程。这个过程可以通过具体实例来说明。假设指数和尾数的宽度分别为2个和1个数字。为了便于理解，我们使用十进制格式。几个有代表性的实例计算如图3-9所示。数值

的计算通过 4 个主要步骤完成：

		排序	对齐	加/减	规范化
示例.1	+0.54E3	−0.87E4	−0.87E4	−0.87E4	−0.87E4
	−0.87E4	+0.54E3	+0.05E4	+0.05E4	+0.05E4
				−0.82E4	−0.82E4
示例.2	+0.54E3	−0.55E3	−0.55E3	−0.55E3	−0.55E3
	−0.55E3	+0.54E3	+0.54E3	+0.54E3	+0.54E3
				−0.01E3	−0.10E2
示例.3	+0.54E0	−0.55E0	−0.55E0	−0.55E0	−0.55E0
	−0.55E0	+0.54E0	+0.54E0	+0.54E0	+0.54E0
				−0.01E0	−0.00E0
示例.4	+0.56E3	+0.56E3	+0.56E3	+0.56E3	+0.56E3
	+0.52E3	+0.52E3	+0.52E3	+0.52E3	+0.52E3
				+1.07E3	+0.10E4

图 3-9　浮点数加法实例

1）排序：把数值较大的数字放在顶部，数值较小的数字放在底部（我们称排序后的数为“大数”和“小数”）；

2）对齐：对齐两个数字，使它们具有相同的指数。可以通过调整“小数”的指数去匹配“大数”的指数来做到。根据指数的差异右移小数字的尾数；

3）加/减：加上或减去两个对齐的数字的有效数字；

4）规范化：调整结果到规范格式。有三种类型的规范化过程：

- 做完减法运算后，结果可能包含前面的数为 0 的情况，如图 3-9 中示例 2 所示；
- 做完减法运算后，结果可能太小而无法被规范化的情况，这样就需要被转换为零，如图 3-9 中示例 3 所示；
- 做完加法运算后，结果可能会产生进位，如图 3-9 中示例 4 所示。

二进制浮点加法器的设计采用了类似的算法。为了简化过程，我们忽略了舍入，在对齐和规范化过程中，当有效位数的低比特位被移出时将被丢弃。设计分为 4 个阶段，每个阶段对应上述算法中的步骤。信号名称后缀中使用的'b'、's'、'a'、'r'、'n'，分别表示“大数”、“小数”、“对齐的数”、“加/减法结果”以及“规范化的数”。根据这些阶段来设计代码，如示例 3. 21 所示。

示例3.21　简单浮点数加法器

```
module fp_adder
    (
    input wire sign1, sign2,
    input wire [3:0] exp1, exp2,
    input wire [7:0]frac1, frac2,
    output reg sign_out,
    output reg [3:0] exp_out,
    output reg [7:0] frac_out
    );
    // 信号定义
    // 信号名称后缀中使用的'b','s','a','n'分别表示
    // "大数"、"小数"、"对齐的数"以及"规范化的数"
    regsign b, signs;
    reg [3:0] expb, exps, expn, exp_diff;
    reg [7:0] fracb, fracs, fraca, fracn, sum_norm;
    reg [8:0] sum;
    reg [2:0] lead0;
    // 实体
    always @ *
    begin
    // 第一步:排序找到大数
    if ({exp1,frac1} > {exp2, frac2})
    begin
    signb = sign1;
    signs = sign2;
    expb = exp1;
    exps = exp2;
    fracb = frac1;
    fracs = frac2;
    end
    else
    begin
    signb = sign2;
```

```
signs = sign1;
expb = exp2;
exps = exp1;
fracb = frac2;
fracs = frac1;
end
//第二步:对齐小数
exp_diff  = expb_exps;
fraca = fracs >> exp_diff;
//第三步:加/减
if (signb == signs)
sum = {1'b0,fracb} + {1'b0, fraca);
el se
sum = {1'b0,fracb) - {1'b0, fraca};
//第四步:格式化
//查高位0 的个数
if (sum[7])
lead0 =3'o0;
else if (sum[6])
lead0 =3'o1;
else if (sum[5])
lead0 =3'o2;
else if (sum[4])
lead0 =3'o3;
else if (sum[3])
lead0 =3'o4;
else if (sum[2])
lead0 =3'o5;
else if (sum[1])
lead0 =3'o6;
else
lead0 =3'o7;
//根据高位0 移位
sum_norm = sum << lead0;
//特殊情况下格式化
```

```
    if (sum[8]) // 带进位;将frac 移至右侧
    begin
      expn = expb + 1;
      fracn = sum[8:1];
    end
    else if (lead0 > expb) // 太小不需格式化
        begin
      expn =0; // 置为0
      fracn =0;
    end
    else
    begin
      expn = expb - lead0;
      fracn = sum_norm;
    end
    // 形成输出
    sign_out = signb;
    exp_out = expn;
    frac_out = fracn;
    end
endmodule
```

电路在第一阶段是比较幅值，并将“大数”发送至信号 signb、expb 和 fracb，将“小数”发送至信号 signs、exps 和 fracs。数值的比较在 expl&frac1 和 exp2&frac2 之间完成。这意味着指数是最先进行比较的，如果相同，再进行底数的比较。

电路在第二阶段进行对齐操作。它首先计算两个指数之间的差异，即 expb-exps，然后将底数 fracs 向右移位 expb-exps 位。对齐的底数被标记为 fraca。电路在第三阶段执行幅值的加法，加法运算与 3.9.2 节相似。注意，操作数被扩展了 1 位，用于放置进位。

电路在第四阶段执行规范化操作，调整结果使最终的输出与规范格式相一致。规范化电路由三段代码构成。第一段计算起始为 0 的个数。有点像一个优先编码器。第二段对底数进行左移操作，移位位数由起始为 0 的计数电路确定。最后一段检查进位和置 0 条件，并生成最终的规范化数。

测试电路　浮点加法器有两个 13 位的输入操作数。由于原型板只有 8 位开

关和 4 个 1 位按钮，无法提供足够数量的物理输入来测试这个电路。为了提供浮点加法器的 26 位输入，我们必须创造一个测试电路，将常量或复制的开关信号赋予加法器的输入操作数。具体的例子如示例 3.22 所示。给一个操作数赋值常量，另一个使用复制的开关信号。加法结果传递到十六进制译码器和符号电路中，并在七段 LED 数码管上显示。

示例 3.22　浮点加法器测试电路

```
module fp_adder_test
    (
    input wire clk,
    input wire [1:0] btn,
    input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
    );
    // 信号定义
    wire sign1, sign2, sign_out;
    wire [3:0] exp1, exp2, exp_out;
    wire [7: 0] frac1, frac2, frac_out;
    wire [7: 0] led3, led2, led1, led0;
    // 实体
    // 建立 fp 地址输入信号
    assign sign1 = 1'b0;
    assign exp1 = 4'b1000;
    assign frac1 = {1'b1, sw[1:0], 5'b10101};
    assign sign2 = sw[7];
    assign exp2 = btn;
    assign frac2 = {1'b1, sw[6:0]};
    //fp 地址例化
    fp_adder_  fp_unit
        (.sign1(sign1),.sign2(sign2),.exp1(exp1),.exp2(exp2),
    .frac1(frac1),.Frac2(frac2),.sign_out(sign_out),
    .exp_out(exp_out),.frac_out(frac_out));
    //3 个16 进制译码器元件的实例化
    hex_to_sseg  sseg_unit_0
```

```
        (. hex(exp_out),. dp(1'b0),. sseg(led0));
    //4LSBs 的部分
    hex_to_sseg   sseg_unit_1
        (. hex(frac_out[3:0]),. dp(1'b1),. sseg(led1));
    //4MSBs 的部分
    hex_to_sseg   sseg_unit_2
        (. hex(frac_out[7:4]),. dp(1'b0),. sseg(led2));
    // 符号位
    assign   led3 = (sign_out)? 8'b11111110:8'b11111111
    // 实例化七段LED 显示时分复用模块
    Disp_mux   disp_unit
        (. clk(clk),. reset(1'b0),. in0(led0),. in1(led1),
    . in2(led2),. in3(led3),. an(an),. sseg(sseg));
endmodule
```

3.10　文献备注

S. Palnitkar 编著的《Verilog HDL, 2nd edition》和 M. D. Ciletti 编著的《Starter's Guide to Verilog2001》对 Verilog 语言的语法和结构进行了详细的说明。S. Sutherland 的文章《The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need It》对新特性进行了总结。C. E. Cummings 的文章《"full-case parallel-case", the Evil Twins of Verilog Synthesis》对 full-case 和 parallel-case 语句使用过程中的注意事项进行了讨论，他的另一篇文章 "New Verilog-2001 Techniques for Creating Parameterized Models" 讨论了 Verilog-2001 新的参数传递机制的优势。

3.11　实验

3.11.1　多功能桶式移位器

思考一个能够完成循环左移或右移的 8 位移位电路。另外一个 1bit 控制信号 lr，用于指定移位方向。

1）设计一个电路，使用一个循环右移电路，一个循环左移电路和一个二选一多路选择器来选择期望的结果，编写代码；

2）编写一个 testbench 并使用 testbench 进行仿真，验证代码的运行情况；

3）对代码进行综合，并实现到 FPGA 开发板中，然后验证它的运行情况；

4）这个电路也能通过一个带有预反转和后反转的循环右移电路实现，反转电路可将输入进行传递或反转（如假设一个 8 位输入是 a7a6a5a4a3a2a1a0，反转的结果变成了 a0a1a2a3a4a5a6a7），重复步骤 2）~3）步骤；

5）检查报告并比较这两种设计的逻辑单元数量和传输延时；

6）修改代码将电路位宽扩展到 16 位并综合代码，重复步骤 1）~步骤 5）。

7）修改代码将电路位宽扩展到 32 位并综合代码，重复步骤 1）~步骤 5）。

3.11.2　双优先级编码器

双优先级译码器返回最高或次高优先级请求的编码。输入是一个 12 位的请求信号，输出是第一和第二，位宽为 4bit，分别是最高或次高优先级请求的二进制编码。

1）设计这个电路并编写代码；

2）编写一个 testbench 并使用 testbench 进行仿真，验证代码的运行情况；

3）设计一个测试电路，用于在开发板上的七段 LED 上显示输出，编写代码；

4）综合代码得到电路，加载到 FPGA 中，验证它的运行情况。

3.11.3　BCD 码增量器

二进制编码的十进制（BCD）格式用 4 位二进制数表示十进制数。例如，259_{10}用二进制编码的十进制（BCD）格式表示为“0010 0101 1001”。BCD 增量器使用 BCD 格式对一个数加 1。例如，增加后，“0010 0101 1001”（也就是，259_{10}）变为“0010 0110 0000”（也就是，260_{10}）。

1）设计一个 3 个数字的 12 位增量器并编写代码；

2）编写一个 testbench 并使用 testbench 进行仿真，验证代码的运行情况；

3）设计一个测试电路用来在七段 LED 上显示这 3 个数字，并编写代码；

4）综合代码得到电路，加载到 FPGA 中，验证它的运行情况。

3.11.4　浮点数大于比较电路

一个浮点数比较电路比较两个浮点数，当第一个数大于第二个数时置位输出信号 gt。假设两个数据格式使用 3.9.4 节所讨论的格式表示。

1）设计这个电路并编写代码；

2）编写一个 testbench 并使用 testbench 进行仿真，验证代码的运行情况；

3）设计一个测试电路并编写代码；

4）综合代码得到电路，加载到 FPGA 中，验证它的运行情况。

3.11.5　浮点数和有符号整型数转换电路

在一个大的系统中，数据可能需要转换到不同的格式。例如我们在3.9.4节中使用的用于表示浮点数的13位格式和用于表示整型数的8位有符号数据类型。整型数到浮点数的转换电路将一个输入的8位整型数转换为规范格式的13位浮点数输出。浮点数转换到整型数的转换电路操作相反。由于浮点数的表示范围非常大，转换可能导致数据下溢的情况（也就是转换后的数的幅值比“00000001”要小）或者上溢的情况（也就是转换后的数据的幅值大于“01111111”）。

1）设计一个整型数到浮点数的转换电路并编写代码；

2）编写一个testbench并使用testbench进行仿真，验证代码的运行情况；

3）设计一个测试电路并编写代码；

4）综合代码得到电路，加载到FPGA中，验证它的运行情况；

5）设计一个浮点数到整型数的转换电路。除8位的整型数输出外，这个设计还应该包括2个状态信号uf和of用来表示下溢和上溢状态。编写代码并重复步骤2）~步骤4）。

3.11.6　增强型浮点型加法器

3.9.4节中的浮点型加法器在低位移出（认为是恢复为0）时，丢弃了低位。一个更精确的方法是使用就近舍入的方式，像IEEE标准中定义的二进制浮点数算术一样（IEEE-754标准）。这一方法的实现需要3个额外的位，分别称之为guard、round和sticky。如果之前学过浮点数的算法，那么可以修改3.9.4节中的浮点数加法器，使其适用就近舍入方法。

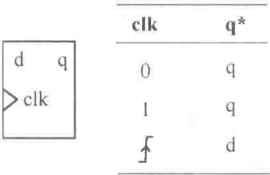
第 4 章　常规时序电路

4.1　简介

时序电路是带有存储器的电路，这些存储器能够产生电路的内部状态，时序电路与组合电路的差别是，组合电路的输出仅与输入有关，而时序电路的输出不仅与输入有关，而且还与内部的状态有关。同步设计方法学是设计时序电路最常用的方法。在同步设计方法学中，所有的存储元件都被一个全局时钟信号控制（同步），并且数据的采样和存储都在这个全局时钟的上升沿和下降沿上进行。这就使得设计者们能够把存储器件从电路中分离出来，从而大大简化开发过程。同步设计方法学是开发大规模复杂数字系统中最为重要的原则，同时也是绝大多数综合、验证和测试算法的基础。本书中所有的设计都遵循同步设计方法学。

4.1.1　D 触发器和寄存器

时序电路中最基本的存储器件是 D 触发器（DFF）。上升沿触发 D 触发器的表示符号和功能表如图 4-1a 所示。D 信号的值将在 clk 信号的上升沿被采样并存



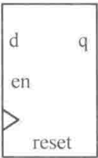
clk	q*
0	q
1	q
↑	d

a) DFF

d q
clk
reset

reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↑	d

b) 带同步复位的DFF



reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↑	0	q
0	↑	1	d

c) 带同步使能的DFF

图 4-1　DFF 的图示和真值表

储到触发器中。D 触发器可能会包含一个异步复位信号，用于将触发器清零。这样的带异步复位端口的 D 触发器的表示符号和功能表如图 4-1b 所示。

DFF 主要有 3 个时间参数，分别是：Tcq（clk 到 q 端的延迟）、Tsetup（建立时间）和 Thold（保持时间）。Tcq 表示的是在时钟上升沿信号从 d 端传播送到 q 端所需要的时间，为了避免触发器进入亚稳态，d 信号必须在采样时钟沿附近处于稳定状态。Tsetup 和 Thold 分别指定了数据信号距离采样沿前后的稳定时间间距。

一个 D 触发器能够提供 1bit 数据的存储，若干个 D 触发器可以组合在一起，用于存储多比特数据，称之为寄存器。

4.1.2　同步系统

图解：图 4-2 是同步系统的框图，包括以下几部分：

- 状态寄存器：由同一个时钟信号控制的 D 触发器集合；
- 次态逻辑：利用外部输入和内部状态（如寄存器的输出）来决定寄存器新值的组合逻辑；
- 输出逻辑：产生的输出信号的组合逻辑。

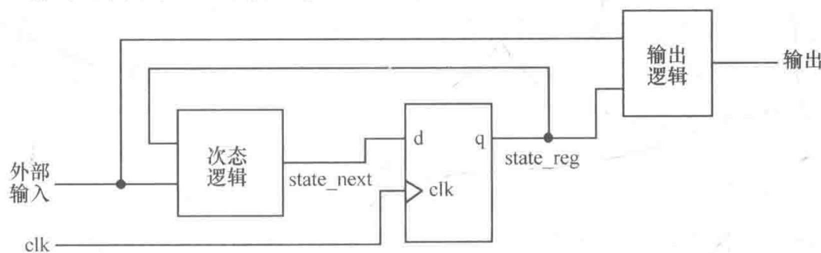


图 4-2　同步时序电路的图示

最高运行频率　时序电路的设计难点之一是确保系统时序没有违反建立保持时间的约束。在一个同步系统中，存储元件如图 4-2 那样被集合起来，看作一个寄存器。我们须对单个存储器件进行时序分析。

时序电路最具代表性的时序参数是最大时钟频率 f_{max}，它是电路最快运行速率的指标。f_{max} 的倒数是最小时钟周期 T_{clock}，即两次采样时钟沿之间的时间间隔。为了确保运行的正确性，下一状态值必须在最小时钟周期内产生并稳定下来。假设下一状态逻辑的最大传输延迟为 T_{comb}，则最小时钟周期可以通过累加闭环电路的传播延迟和建立时间约束获得，如图 4-2 所示。

$$T_{clock} = T_{cq} + T_{comb} + T_{setup}$$

它的倒数即为最大时钟频率：

$$f_{max} = \frac{1}{T_{clock}} = \frac{1}{T_{cq} + T_{comb} + T_{setup}}$$

Xilinx 时序约束在综合过程中，Xilinx 软件会对综合后的电路进行分析并在

综合报告中报告f_{max}指标。我们可以设置期望的运行频率作为综合约束，综合软件将会尝试构造满足频率要求（即f_{max}等于或者远大于期望运行频率）的电路。例如，如果我们在开发板上使用 50MHz（周期 20ns）的晶振作为时钟源，那么时序电路时钟的最大频率f_{max}必须超过 50MHz，（周期必须比 20ns 小）。可以在时序约束文件中加上以下几句：

```
NET "clk" TNM_NET = "clk" ;
TIMESPEC "TS_clk" = PERIOD "clk" 20 ns HIGH 50% ;
```

它表示这个时钟信号有 20ns 的周期（50MHz）以及 50% 的占空比。

综合以后，我们可以调用 ISE 软件“Processes”窗口的 View Design Summary 查看相关的时序信息。其中，Timing Constraints 部分展示了所施加的约束是否得到满足，Static Timing Report 部分提供了更加详细的时序信息。

4.1.3 代码开发

我们的代码开发遵循了图 4-2 所示的基本框图。关键是将存储器件（比如寄存器）从系统分隔出来。一旦寄存器被孤立出来，余下的部分便是一个纯组合电路，前面章节讨论的代码和分析方法就可以得到相应的应用了。虽然这个方法可能有时会让编码变得有点复杂，但却能够帮助我们更好地呈现电路结构，避免一些无意识的存储和一些微小的错误。

根据次态逻辑的特点，我们把时序电路划分为三类：

- 常规时序电路：在这类电路中，状态转移呈现“有规则”的变化模式，如计数器、移位寄存器等，次态逻辑主要通过一个预先设计的、“有规则”的元件构建，如增量器或者移位器；
- FSM：这类电路的状态转移通常不会呈现一种简单、重复的模式，次态逻辑由“随机逻辑”来构造，并从无到有开始综合，应该称之为随机时序电路，但通常称之为 FSM（有限状态机）；
- FSMD：电路由常规时序电路和 FSM 两部分构成，这两部分分别被称为数据路径和控制路径，完整的电路被称为 FSMD（包含有数据路径的 FSM），这类电路一般用于使用寄存器传输（RT）方法学实现某种算法，寄存器传输（RT）方法学通过一系列的数据传输和对寄存器的控制描述系统的运行；
- 这三类电路将在本章以及后两章中进行介绍。

4.2 触发器和寄存器的 HDL 代码

用 HDL 描述存储器件是一个很小的过程，可以通过很多途径实现。实际上，菜鸟 HDL 设计师最常遇到的一个问题是产生非预期的锁存器和缓冲器。我们对

一些常用的存储元件提供了代码模板，用于代替覆盖语意的可能形式。由于开发过程将寄存器和组合逻辑电路进行了分离，本书中这些元件对于设计来说已经足够用了。这些元件是：

- 触发器；
- 寄存器；
- 寄存器文件。

所有的代码模板都是使用阻塞赋值。如3.3.2节描述的那样，描述存储元件应使用非阻塞赋值，非阻塞赋值的语法如下：

[variable-name] <= [expression];

这种赋值方式能够避免一些“竞争”问题和消除仿真与综合之间的差异。非阻塞赋值的知识在本书7.1节中将会有具体介绍。

4.2.1 D触发器

D触发器有三种类型：

- 不带异步复位端的D触发器；
- 带异步复位端的D触发器；
- 带同步使能端的D触发器。

前两种类型的D触发器是最基本的存储元件，并且在任何器件工艺库中都可以找到。第三种D触发器能通过简单的D触发器构造而成。本书包含了相关代码，因为它是频繁使用的存储元件，且能被映射到Spartan-3逻辑单元的触发器上。

不带异步复位端的D触发器　触发器的功能表如图4-1a所示，代码描述见示例4.1。

示例4.1　不带异步复位端的D触发器

```
module d_ff
   (
   input wire clk,
   input wire d,
   output reg q
   );
   // 实体
   always @ (posedge clk)
   q <= d;
endmodule
```

在敏感列表中时钟上升沿用 posedge 来表示。关键词 posedge（即 positive edge）表明了时钟信号向电平 1 变化，表示 always 块只在时钟信号上升沿才被执行。我们注意到 d 信号没有在敏感信号列表里出现，这就说明了：d 信号只能在时钟上升沿到来时被采样，如果时钟上升沿没有到来，d 值的变化不会引发任何立即响应。

带异步复位的 D 触发器　D 触发器可能带有一个异步复位信号端，如图 4-1b 所示的功能表。异步复位信号能够在任何时刻把触发器清零，而不受时钟信号的控制。异步复位信号通常比其他采样输入端的优先级要高。使用异步复位信号违反了同步设计方法学，因此应避免在正常操作中使用。异步复位主要应用是实现系统初始化。例如，可通过产生一个短时复位脉冲强制系统上电后进入到初始状态。示例 4.2 所示代码是带有异步复位端的 D 触发器。

示例 4.2　带异步复位的触发器

```
module d_ff_reset
   (
      input wire clk, reset,
      input wire d,
      output reg q
   );
      // 实体
      always@ (posedge clk, posedge reset)
      if(reset)
         q <= 1'b0;
      else
         q <= d;
endmodule
```

注意，上升沿的复位事件也被包含在了敏感列表中，并且在 if 语句中首先检查了复位信号 reset 的值。如果复位信号被置为有效，则信号 q 会被清零，且该动作不会受到时钟信号的制约。

带同步使能的 D 触发器　D 触发器可能还会带有一个控制信号 en，用于使能触发器采样输入值。其符号表示和功能表如图 4-1c 所示。注意，en 信号只会在时钟的上升沿被检验，因此是同步的。如果没有被置为有效，那么触发器将保持原值。代码如示例 4.3 所示。

示例 4.3　带同步使能的 D 触发器代码风格

```
module d_ff_en_lseg
    (
        input wire clk, reset,
        input wire en,
        input wire d,
        output reg q
    );
        // 实体
        always @ posedge clk, posedge rest)
        if (reset)
            q <= 1'b0;
        else if (en)
            q <= d;
    endmodule
```

注意，第二个 if 语句之后没有 else 分支。根据 Verilog 语法定义，如果一个变量没有被赋值那么它将保持之前的值。如果 en 为 0，那么 q 保持它的之前值。因此，漏掉了 else 分支恰好描述了触发器的行为。

带使能端的 DFF 对于保持一个较高速的子系统和一个较低速的子系统之间的同步性，是非常有用的。例如，假设一个高速子系统和一个低速子系统的工作频率分别为 50MHz 和 1MHz，可以利用每隔 50 个时钟周期产生的一个周期性使能信号来驱动 1MHz 的低速子系统，而不是使用分频派生的 1MHz 时钟。在剩余的 49 个时钟周期内这个低速的子系统是不工作的（即保持以前的状态）。同样的策略还可用来去除门控时钟。

由于使能信号是同步的，电路可以通过常规的 D 触发器和简单的次态逻辑构造。示例 4.4 为 D 触发器的代码，图 4-3 为它的框图。

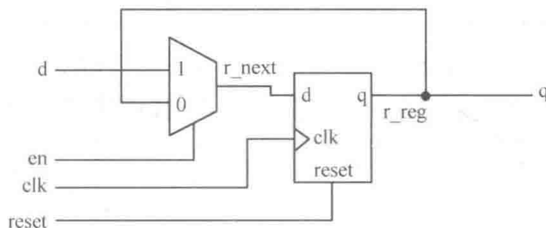


图 4-3　带同步使能的 D 触发器

示例 4.4　带同步使能端的 DFF 第二种代码表示

```
module d_ff_en_2seg
    (
    input wire clk, reset,
    input wire en,
    input wire d,
    output reg q
    );
    // 信号的声明
    reg r_reg, r_next;
    // 实体
    //D FF
    always @ (posedge clk, posedge reset)
        if (reset)
            r_reg <= 1'b0;
        else
            r_reg <= r_next;
    // 次态逻辑
    always @ *
        if (en)
            r_next = d;
        else
            r_next = r_reg;
    // 输出逻辑
    always @ *
        q = r_reg;
endmodule
```

为了清晰起见，我们使用后缀_next 以及_reg 来强调下一次输入的值以及触发器的寄存输出值，它们分别是 D 触发器的 d 端信号以及 q 端信号。可以把示例 4.3 看作是这段复杂描述的速记。

4.2.2　寄存器

一个寄存器是由同一时钟和复位信号控制的 D 触发器的集合，与 D 触发器

一样，寄存器拥有一个可供选择的异步复位信号以及同步使能信号。除了相关输入/输出信号需要使用数组类以外，寄存器的代码与 D 触发器的代码描述是一样的。如示例 4.5 中描述的带异步复位功能的 8bit 寄存器。

示例 4.5　寄存器

```
module reg_reset
    (
    input wire clk, reset,
    input wire [7:0] d,
    output reg [7:0] q
    );
    // 实体
    always @ (posedge clk, posedge reset)
        if (reset)
            q <=0;
        else
            q <=d;
endmodule
```

4.2.3　寄存器文件

寄存器文件是带有一个输入端口和一个或一个以上输出端口的寄存器集合，写地址信号 w_addr 指定了存储数据的地址，而读地址信号 r_addr 则指定了要获取数据的地址。寄存器文件一般用于快速的、临时性的存储。示例 4.6 为 2W × B 的寄存器文件代码。两个在设计中定义的参数：W 用于指定地址的位宽，暗含寄存器文件可存储 2W 个 word，B 用于指定每一个 word 的位宽为 B。

示例 4.6　寄存器文件代码

```
module reg_file
    #(
    parameter B =8, // 比特数
              W =2  // 地址比特数
    )
    (
    input wire clk,
```

```
    input wire wr_en,
    input wire [W-1:0] w_addr, r_addr,
    input wire [B-1:0] w_data,
    output wire [B-1:0] r_data
    );
    // 信号的声明
    reg [B-1:0] array_reg [2**W-1:0];
    // 实体
    // 写操作
    always @(posedge clk)
        if (wr_en)
            array_reg[w_addr] <= w_data;
    // 读操作
    assign r_data = array_reg[r_addr];
endmodule
```

这段代码包含了一些新特性。首先，定义了一个二维数组，即

reg [B-1:0] array_reg [2**W-1:0];

它表示变量 array_reg 是有[2**W-1:0]元素的数组，每个元素中数据类型为 reg[B-1:0]。第二，可以用一个索引信号访问数组中的某个元素，例如 array_reg [w_addr]。尽管描述很抽象，但 Xilinx 软件能够识别这种语言结构，并能获到正确的实现。array_reg[…] = …和… = array_reg[…]语句能够推断出相应的译码和多路选择逻辑。

一些应用可能需要在同一时间获取多个数据。可以通过增加一个额外的读端口实现：

r_data2 = array_reg [r_addr_2];

4.2.4 Xilinx Spartan-3 器件的存储元件

在 Xilinx Spartan-3 器件中，每个逻辑单元都含有带异步复位端和同步使能端的 D 触发器，这些 D 触发器基本上构成了图 4-2 所示的寄存器。由于逻辑单元还包含了一个四输入的 LUT，因此如果一个逻辑单元仅作为大容量存储器的其中 1 位，是非常浪费资源的。Spartan-3 器件还带有分布式 RAM（随机存储器）以及固定的 RAM 模块，一般用于满足较大的存储需求。这些存储模块可以配置为同步操作模式，其特点类似于一个寄存器文件的受限制版本。这些模块的配置和推断将在第 12 章中讨论。

4.3　简单的设计举例

我们将在这一章节中介绍一些简单的、有代表性的时序电路构造方法。

4.3.1　移位寄存器

Free_running 移位寄存器　Free_running 移位寄存器在每一个时钟周期内将寄存器中的内容向左或者向右移动一位。没有其他的控制信号。*N*bit 的 free-running 移位寄存器的代码如示例 4.7 所示。

示例 4.7　Free-running 移位寄存器

```
module free_run_shift_reg
    #(parameter N = 8)
    (
    input wire clk, reset,
    input wire s_in,
    output wire s_out
    );
    // 信号的声明
    reg [N-1:0] r_reg;
    wire [N-1:0] r_next;
    // 实体
    // 寄存器
    always @ (posedge clk, posedge reset)
        if (reset)
            r_reg <= 0;
        else
            r_reg <= r_next;
    // 次态逻辑
    assign r_next = {s_in, r_reg[N-1:1]};
    // 输出逻辑
    assign s_out = r_reg[0];
endmodule
```

次态逻辑是一个 1bit 的移位器，它将 r_reg 右移一位，并将串行输入信号 s_

in 插入到最高位（MSB）。由于 1bit 移位器仅是将输入和输出重新连接，因此不需要真正的逻辑。它的传播延时代表最小可能延时 T_{comb}，对应的 f_{max} 代表当前器件工艺所能达到的最高时钟速率。

通用移位寄存器　通用移位寄存器可以装载并行的数据，向左或向右移动内容，或者保留同样的状态。它可以完成并串转换功能（先装载并行输入然后移位），或者完成串并转换功能（先移位然后得到并行输出）。期望的操作可通过 2bit 控制信号（ctrl）确定。代码实现如示例 4.8 所示。

示例 4.8　通用移位寄存器

```
module univ_shift_reg
   #(parameter N = 8)
   (
   input wire clk, reset,
   input wire [1:0] ctrl,
   input wire [N-1:0] d,
   output wire [N-1:0] q
   );
   // 信号的声明
   reg [N-1:0] r_reg, r_next;
   // 实体
   // 寄存器
   always @ (posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;
      else
         r_reg <= r_next;
   // 次态逻辑
   always @ *
     case(ctrl)
       2'b00: r_next = r_reg;                          // 无操作
       2'b01: r_next = {r_reg[N-2:0], d[0]};           // 左移
       2'b10: r_next = {d[N-1], r_reg[N-1:1]};         // 右移
       default: r_next = d;                            // 读取
     endcase
   // 输出逻辑
```

```
    assign q = r_reg;
endmodule
```

次态逻辑使用了一个4选1的多路选择器来选择期望的寄存器新值。注意，左移或右移操作中，d的最高位和最低位（即d[0]和d[N-1]）被用于串行输入。

在Xilinx Spartan3器件中，逻辑单元的4输入LUT是由16×1的SRAM来实现的，同样的SRAM还可以配置成16个1bit SRAM Xilinx单元级联链，就像16bit的移位寄存器。这可以用来构造确定形式的移位寄存器并且能够使操作有效可行。

4.3.2 二进制计数器及其转换形式

Free_running二进制计数器 Free_running二进制计数器是通过不断重复二进制序列来实现计数的。例如，一个4bit的二进制计数器从“0000”，“0001”，……到“1111”计数并且不断循环。示例4.9表示的是一个*N*bit参数化的二进制计数器。

示例4.9 Free-running二进制计数器

```
module free_run_bin_counter
    #(parameter N = 8)
    (
    input wire clk, reset,
    output wire max_tick,
    output wire [N-1:0] q
    );
    // 信号的声明
    reg [N-1:0] r_reg;
    wire [N-1:0] r_next;
    // 实体
    // 寄存器
    always @(posedge clk, posedge reset)
        if (reset)
            r_reg <= 0;  // {N{1b'0}}
        else
            r_reg <= r_next;
```

```
    // 次态逻辑
    assign r_next = r_reg + 1;
    // 输出逻辑
    assign q = r_reg;
    assign max_tick = (r_reg == 2 * * N - 1) ? 1'b1 : 1'b0;
      // 也可以用 (r_reg == {N {1 'b1 } } )
endmodule
```

次态逻辑是一个对当前寄存器值加 1 的增量器。由 + 操作符定义可知，r_reg 计数到"1……1"后，加法器重新由 0 开始计算，该电路还含有一个输出状态信号 max_tick，当计数器加到值最大值"1……1"（即计数到 2^N-1）时，max_tick 置为有效。

max_tick 信号是一种特定类型的信号，它只有一个时钟周期的有效时间。本书中，我们称之为"tick"，并且利用后缀名"_tick"来标注这种类型的信号。通常用作与其他时序电路的接口的使能信号。

通用二进制计数器　通用二进制计数器更加便于使用，它能够向上或向下计数，也可以暂停，还可以载入某个特殊值，也可以被同步清零。表 4-1 总结了通用二进制计数器的功能。注意 reset 信号和 syn_clr 信号之间的区别性，前者是异步的，并且只能用于系统初始化，而后者在时钟的上升沿被采样，能够应用于一般的同步设计中。计数器代码如示例 4.10 所示。

表 4-1　通用二进制计数器的功能表

syn_clr	load	en	up	q *	操作
1	—	—	—	00…00	同步清零
0	1	—	—	d	并行加载
0	0	1	1	q + 1	向上计数
0	0	1	0	q - 1	向下计数
0	0	0	—	q	暂停

示例 4.10　通用二进制计数器

```
module univ_bin_counter
    #(parameter N = 8)
    (
    input wire clk, reset,
    input wire syn_clr, load, en, up,
```

```
    input wire [N-1:0] d,
    output wire max_tick, min_tick,
    output wire [N-1:0] q
    );
    // 信号的声明
    reg [N-1:0] r_reg, r_next;
    // 实体
    // 寄存器
    always @(posedge clk, posedge reset)
        if(reset)
            r_reg <= 0;    //
        else
            r_reg <= r_next;
    // 次态逻辑
    always @ *
        if (syn_clr)
            r_next = 0;
        else if (load)
            r_next = d;
        else if (en & up)
            r_next = r_reg + 1;
        else if (en & ~up)
            r_next = r_reg - 1;
        else
            r_next = r_reg;
    // 输出逻辑
    assign q = r_reg;
    assign max_tick = (r_reg == 2 ** N - 1) ? 1'b1 : 1'b0;
    assign min_tick = (r_reg == 0) ? 1'b1 : 1'b0;
endmodule
```

always 模块中描述了次态逻辑在功能表中所述的功能，在 always 模块中使用了 if 语句来优先处理期望操作。

模 m 计数器　模 m 计数器能够从 0 到 m－1 不断循环计数，示例 4.11 是描述模 m 计数器的代码。它有两个参数：M 指定了计数的范围 m，N 指定了需要

的比特位数，应等于 $\lceil \log_2 M \rceil$，代码见示例4.11，默认值为模10计数器。

示例4.11　模 m 计数器

```
module mod_m_counter
   #(
   parameter N = 4,  // 计数器中的比特数
             M = 10  // 模m
   )
   (
   input wire clk, reset,
   output wire max_tick,
   output wire [N-1:0] q
   );
   // 信号的声明
   reg [N-1:0] r_reg;
   wire [N-1:0] r_next;
   // 实体
   // 寄存器
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;
      else
         r_reg <= r_next;
      // 次态逻辑
   assign r_next = (r_reg == (M-1)) ? 0 : r_reg + 1;
   // 输出逻辑
   assign q = r_reg;
   assign max_tick = (r_reg == (M-1)) ? 1'b1 : 1'b0;
endmodule
```

次态值由条件操作符构造。如果计数值到 M-1，新值就会清零，否则加1。

由于 N 的值取决于 M 的值，所以代码中另外定义的参数 N 多少还是有些多余。一个更好的办法是定义一个能根据 M 值自动计算 N 值的函数。该方法会在7.4节讨论。

4.4　时序电路的测试平台

如1.7节所述，测试平台是一个用于模拟物理实验平台的程序。本节中，我们将介绍如何去构造简单的通用二进制计数器的测试平台，还可以当作测试其他时序电路的模板。本书会在7.5节中介绍如何构建更加复杂的测试平台。简单的测试平台构造如示例4.12所示。

示例4.12　通用二进制计数器的测试平台

```
timescale 1 ns/10 ps
//timescale 命令说明
//仿真的时间单位是1ns
//仿真器
module bin_counter_tb();
   //声明
   localparam T=20; //时钟周期
   reg clk, reset;
   reg syn_clr, load, en, up;
   reg [2:0] d;
   wire max_tick, min_tick;
   wire [2:0] q;
   //被测单元例化
   univ_bin_counter #(.N(3))uut
      (.clk(clk),.reset(reset),.syn_clr(syn_clr),
       .load(load),.en(en),.up(up),.d(d),
       .max_tick(max_tick),.min_tick(min_tick),.q(q));
   //时钟
   //时钟周期始终为20ns
   always
   begin
      clk=1'b1;
      #(T/2);
      clk=1'b0;
      #(T/2);
   end
```

```
在最初的半个时钟周期生成复位信号
initial
begin
    reset = 1'b1;
    #(T/2);
    reset = 1'b0;
end
// 其他激励
initial
begin
    // ====输入信号初始值 =====
    syn_clr = 1'b0;
    load = 1'b0;
    en = 1'b0;
    up = 1'b1; // 对up 计数
    d = 3'b000;
    @(negedge reset);   // 等待复位解除
    @(negedge clk);     // 等待一个时钟周期
    // ====测试装载操作 =====
    load = 1'b1;
    d = 3'b011;
    @(negedge clk);     // 等待一个时钟周期
    load = 1'b0;
    repeat(2) @(negedge clk);
    // ====测试清除操作 ====
    syn_clr = 1'b1;  // 清除信号有效
    @(negedge clk);
    syn_clr = 1'b0;
    // ====测试上升计数器和暂停操作 ====
    en = 1'b1; // 计数
    up = 1'b1;
    repeat(10) @(negedge clk);
    en = 1'b0; // 暂停
    repeat(2) @(negedge clk);
    en = 1'b1;
```

```
        repeat(2) @ (negedge clk);
        // ====测试下降计数器 ====
        up = 1'b0;
        repeat(10) @ (negedge clk);
        // ====等待语句 ====
        // 等到q =2 时继续
        wait(q ==2);
        @ (negedge clk);
        up = 1'b1;
        // 等到min-tick 变成1 时继续
        @ (negedge clk);
        wait(min_tick);
        @ (negedge clk);
        up = 1'b0;
        // ====绝对时延 ====
        #(4 * T);  // 等待80ns
        en = 1'b0; // 暂停
        #(4 * T);  // 等待80ns
        // ====停止仿真 ====
        // 返回交互仿真模式
        $ stop;
    end
endmodule
```

代码含有一句用于创建一个 3bit 计数器实例的语句，以及三段用于生成时钟（clock），复位（reset）和其他的常用输入的语句。时钟的生成可以由一个 always 块创建：

```
always
begin
  clk = 1'b1;
  #(T/2);
  clk = 1'b0;
  #(T/2);
end
```

T 是一个表示时钟周期的常量，定义如下：

localparam T = 20;
注意，该 always 块没有敏感信号列表，不断进行自我循环。时钟信号被交替赋予 0 或者 1，每一个值保持半个时钟周期。

复位信号激励是通过 initial 块模拟产生的：

```
initial
begin
  reset = 1'b1;
  #(T/2);
  reset = 1'b0;
end
```

initial 块在仿真一开始就立即执行。这表明了复位信号初值被置为 1，半个时钟周期后变为 0。initial 块表示的是上电条件：复位信号即刻对系统进行清理，进入初始状态。注意，默认情况下 x 值（不定态）被赋予一个变量值，而不是 0。使用短时复位脉冲是实现系统初始化的一个不错的机制。

第二个 initial 块生成用于施加到其他输入信号的激励。首先对装载和清除操作进行测试，然后再测试两个不同方向的计数功能。最后的 $ stop 函数用于强制仿真器停止仿真。对于一个使用正触发驱动触发器的系统而言，一个输入信号必须在时钟信号的上升沿附近保持稳定，以满足建立时间和保持时间约束。一个简单方法就是在时钟信号由 1 到 0 跳变的时候改变输入信号值。可通过使用下面语句等待这一变化：

```
@(negedge clk);
```

negedge clk 事件指的是时钟信号跳变到 0 的条件（即下降沿）。注意，每条语句都代表一个新的下降沿，对应着一个新的时钟周期的推进。在我们的模板中，我们通常使用这种语句来指定时序过程。对于多个时钟周期，可以使用 repeat 语句，如

```
repeat (10) @(negedge clk); // 循环10 个时钟周期;
```

一些额外的时序控制结构放在了 initial 块的最后。我们可以等待直到某些特殊条件满足。如“当 q 等于 2”，代码如下：

```
wait (q==2);
```

或者等待信号的跳变，如

```
wait (min_tick);
```

又或是等待某个绝对时间，如

```
#(4*T);  // 等待4 个T,
```

若一个输入信号在这些语句后发生改变，则需要保证这些输入信号的跳变不能发生在时钟上升沿上，必要时，我们应增加以下语句：

```
@ (negedge clk);
```

接下来可以编译代码并实施仿真。图4-4是部分仿真的波形。

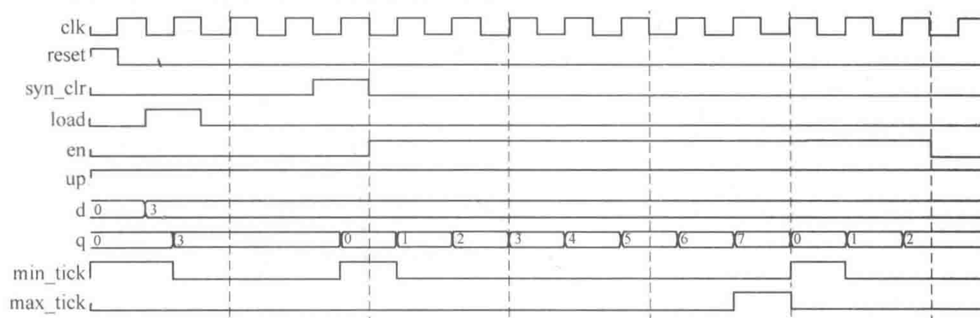


图4-4 测试平台波形图

4.5 案例学习

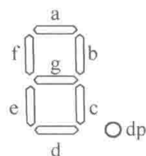
经过一些简单的电路练习之后，这一节会讨论一些更加复杂的设计实例。

4.5.1 LED分时复用电路

S3板上有4个七段数码管，每一个数码管包含7个条段和1个小圆点，为了减少FPGA的I/O引脚的使用，S3板使用了分时复用共享策略。在此策略中，4个七段数码管拥有独立的使能信号，但是共享了8个公共信号，用于亮灯数码管的条段。所有信号低电平有效（即，当信号为0时使能）。数字“3”的显示原理图如图4-5所示。注意，使能信号（即an）为“1110”。这样的配置每次只能使能一个数码管的显示。我可以按图4-6中所示的简化时序图那样通过轮流使能来分时复用这4个数码管。如果使能信号的刷新速率足够快，人眼便无法分辨LED的开关间隔，会在感官上认为这4个灯是同时亮的。这一方法将I/O引脚数从32个降低到12个（即8个LED数码管加4个使能信号）。本书将在下一节中讨论这一电路的两个形式。

使用LED的分时复用电路 分时复用电路的符号和框图如图4-7所示。它使用了4个七段LED的图像，in3、in2、in1和in0，并依照使能信号把它们输出到sseg上。

使能信号的刷新频率必须保持足够快，才能欺骗我们的视觉，但也不能过快，以保证LED能进行完整的开关操作。比较合适的频率是1000Hz左右。在我们的设计中，为了达到这个目的，我们使用一个18位的二进制计数器。高两位被译码产生使能信号，用于多路开关的选择信号。单个比特位的刷新频率，例如第［0］位，为（$50M/2^{16}$）Hz，约800Hz。代码见示例4.13。



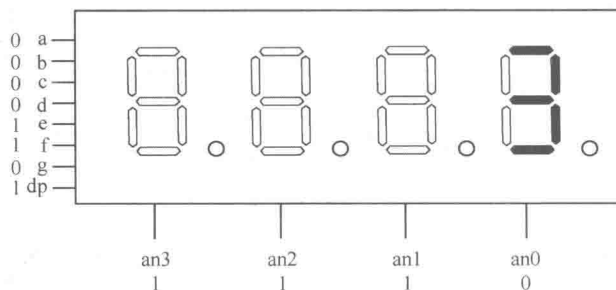


图 4-5　七段数码管的分时复用电路

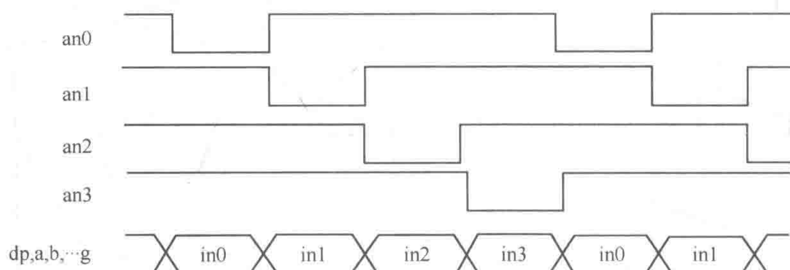


图 4-6　七段数码显示管的分时复用电路时序图

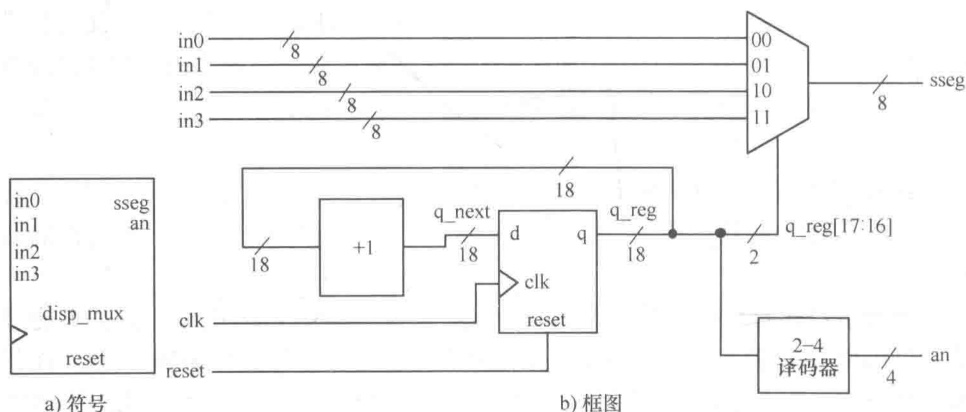


图 4-7　分时复用电路的标识以及电路图

示例4.13 使用LED图样的分时复用电路

```
moduledisp_mux
    (
    input wire clk, reset,
    input [7:0] in3, in2, in1, in0,
    output reg [3:0] an,  // 使能,4 位当4 有1 位被置低
    output reg [7:0] sseg  //LED 段
    );
    // 常量声明
    // 刷新频率约为800Hz (50MHz/2^16)
    localparam N = 18;
    // 信号声明
    reg [N-1:0] q_reg;
    wire [N-1:0] q_next;
    //N 位计数器
    // 寄存器
    always @ (posedge clk, posedge reset)
        if (reset)
            q_reg <= 0;
        else
            q_reg <= q_next;
    // 次态逻辑
    assign q_next = q_reg + 1;
    // 计数器的高两位控制4:1 多路选择器
    // 并且产生低有效使能信号
    always @ *
        case (q_reg[N-1:N-2])
            2'b00:
                begin
                    an = 4'b1110;
                    sseg = in0;
                end
            2'b01:
                begin
```

```
                an =4'b1101;
                sseg = in1;
                end
        2'b10:
            begin
                an =4'b1011;
                sseg = in2;
                end
        default:
            begin
                an =4'b0111;
                sseg = in3;
            end
    endcase
endmodule
```

我们使用图 4-8 的测试电路去验证 LED 分时复用电路的运行情况。测试电路使用了 8bit 的寄存器存储 LED 的显示图样，这些寄存器使用相同的 8bit 开关作为输入，但却由各自的使能信号控制。当我们按下一个按钮，相应的寄存器被使能，显示样式便被加载到这个寄存器中。代码如示例 4. 14 所示。

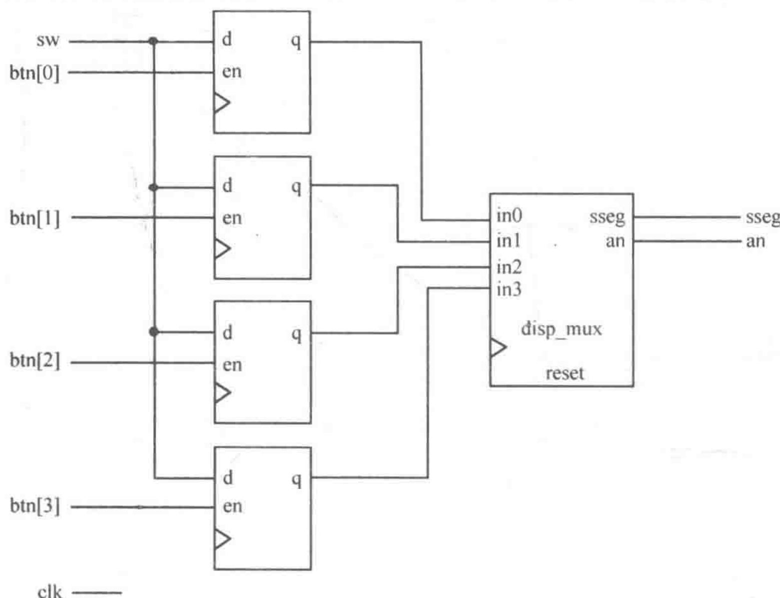


图 4-8　LED 分时复用测试电路

示例 4.14　LED 的分时测试电路

```
module disp_mux_test
   (
   input wire clk,
   input wire [3:0] btn,
   input wire [7:0] sw,
   output wire [3:0] an,
   output wire [7:0] sseg
   );
   // 信号声明
   reg [7:0] d3_reg, d2_reg, d1_reg, d0_reg;
   // 例化7 段数码管分时复用模块
   disp_mux disp_unit
      (.clk(clk),.reset(1'b0),.in0(d0_reg),.in1(d1_reg),
       .in2(d2_reg),.in3(d3_reg),.an(an),.sseg(sseg));
   //4 个数码管显示图样寄存器
   always @ (posedge clk)
   begin
      if (btn[3])
         d3_reg <= sw;
      if (btn[2])
         d2_reg <= sw;
      if (btn[1])
         d1_reg <= sw;
      if (btn[0])
         d0_reg <= sw;
   end
endmodule
```

十六进制数据的并行输入　七段数码显示最通常的应用是用于十六进制数字的显示。其译码电路将在第 3.9.1 节中讨论。为了能够在之前的分时复用电路上显示 4 个十六进制数据，我们需要 4 个译码电路。一个更好的方法是先多路选择十六进制数字，然后对结果进行译码，如图 4-9 所示。

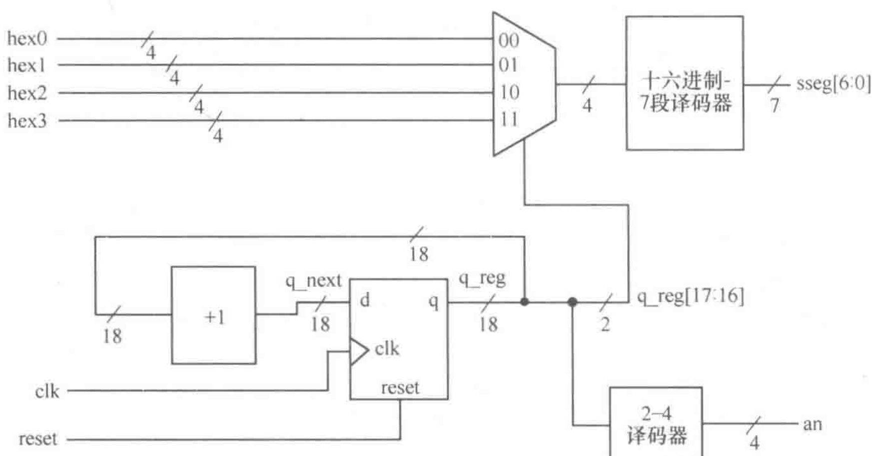


图 4-9　十六进制分时复用电路框图

该方法仅需一个译码电路，并且可以把 4:1 多路选择器的位宽从 8 位减少到 5 位（4 位用于十六进制数字，1 位用于小数字）。对应代码如示例 4. 15 所示。除了时钟和复位，输入信号还包括了 4 个 4 位的十六进制数字，hex3，hex2，hex1 和 hex0，以及 4 个小数点组成的信号 dp_in. 。

示例 4. 15　十六进制数字的 LED 分时复用电路

```
module disp_hex_mux
   (
   input wire clk, reset,
   input wire [3:0] hex3, hex2, hex1, hex0,   十六进制
   input wire [3:0] dp_in,               //4 个小数点
   output reg [3:0] an,  // 使能, 4 位当4 有1 位被置低
   output reg [7:0] sseg //LED 段
   );
   // 常量声明
   // 刷新频率约为800Hz (50MHz/2^16 )
   localparam N = 18;
   // 内部信号声明
   reg [N-1:0] q_reg;
   wire [N-1:0] q_next;
   reg [3:0] hex_in;
   reg dp;
```

```
//N 位计数器
// 寄存器
always @ (posedge clk, posedge reset)
    if (reset)
        q_reg <= 0;
    else
        q_reg <= q_next;
// 次态逻辑
assign q_next = q_reg + 1;
// 计数器的高两位控制4:1 多路选择器
// 并产生低有效使能信号
always @ *
    case (q_reg[N-1:N-2])
        2'b00:
            begin
                an = 4'b1110;
                hex_in = hex0;
                dp = dp_in[0];
            end
        2'b01:
            begin
                an = 4'b1101;
                hex_in = hex1;
                dp = dp_in[1];
            end
        2'b10:
            begin
                an = 4'b1011;
                hex_in = hex2;
                dp = dp_in[2];
            end
        default:
            begin
            an = 4'b0111;
            hex_in = hex3;
```

```
            dp = dp_in[3];
         end
      endcase
   // 十六进制数与7 段数码管的对应
   always @ *
   begin
      case(hex_in)
         4'h0: sseg[6:0] =7'b0000001;
         4'h1: sseg[6:0] =7'b1001111;
         4'h2: sseg[6:0] =7'b0010010;
         4'h3: sseg[6:0] =7'b0000110;
         4'h4: sseg[6:0] =7'b1001100;
         4'h5: sseg[6:0] =7'b0100100;
         4'h6: sseg[6:0] =7'b0100000;
         4'h7: sseg[6:0] =7'b0001111;
         4'h8: sseg[6:0] =7'b0000000;
         4'h9: sseg[6:0] =7'b0000100;
         4'ha: sseg[6:0] =7'b0001000;
         4'hb: sseg[6:0] =7'b1100000;
         4'hc: sseg[6:0] =7'b0110001;
         4'hd: sseg[6:0] =7'b1000010;
         4'he: sseg[6:0] =7'b0110000;
         default: sseg[6:0] =7'b0111000;   //4 'hf
      endcase
      sseg[7] = dp;
   end
endmodule
```

为了验证这个电路的运行，我们使用 8 个开关来表示 2 个 4bit 的无符号数，对两个数相加，并在 4 个七段数码管上显示这两个数以及它们的和。代码如示例 4. 16 所示。

示例 4. 16　十六进制数据的分时测试电路

```
module hex_mux_test
   (
```

```
    input wire clk,
    input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
    );
    // 信号声明
    wire [3:0] a, b;
    wire [7:0] sum;
    // 例在7 段数码管
    disp_hex_mux disp_unit
        (.clk(clk),.reset(1'b0),
         .hex3(sum[7:4]),.hex2(sum[3:0]),.hex1(b),.hex0(a),
         .dp_in(4'b1011),.an(an),.sseg(sseg));
    // 加法器
    assign a = sw[3:0];
    assign b = sw[7:4];
    assign sum = {4'b0,a} + {4'b0,b};
endmodule
```

仿真注意事项　本书中的许多时序电路是在速率相对较低的条件下运行的，如 LED 分时复用电路中的使能脉冲。这种情况可以利用计数器产生一个单周期的使能信号来实现。在本电路中使用了一个 18 位的计数器。

```
localparam N = 18;
reg [N-1:0] q_reg;
wire [N-1: 0] q_next;
...
assign q_next = q_reg + 1;
```

因为计数器的大小，这种电路的仿真会耗费相当数量的计算时间（需要耗费 2^{18} 个时钟周期完成一次迭代）。由于我们的关注点在代码的复用开关部分，因此大部分的仿真时间被浪费了。所以在仿真中使用较小的计数器效率会更高一些。在构造这个 testbench 时，可以通过定义一个常数来实现这一点：

```
    localparam N = 4;
```

仅仅需要 24 个时钟周期就能完成一次迭代，还可以让我们更好地检验和观察关键的运行。

我们可以将 N 定义为一个参数，而不是在仿真和综合之间使用常量或者修

改代码。在实例化过程中，可以分别为仿真和综合赋予不同的值。

4.5.2　码表

这一节我们将讲述码表的设计。码表可显示精度为 3 位十进制数的时间，可从 00.0s 计时到 99.9s 然后从头开始。它包含了一个同步清零信号 clr，可以使计数返回到 00.0，还含有 1 个用来启动或者暂停计数的使能信号 go。这个设计基本上算是个 BCD 码（binary - codede decimal）计数器，以 BCD 码格式计数。在这个格式中，十进制数由一个 4 位的 BCD 码表示。例如，139_{10} 由 “000100111001” 表示，下一个数 140_{10} 由 “0001 0100 0000” 表示。

由于 S3 板有个 50MHz 的时钟，我们首先需要一个模 5，000，000 的计数器，用于在每隔 0.1s 产生一个单周期的使能信号。该使能信号用于使能这个 3 位数 BCD 计数器的计数。

设计 I 我们的第一个 BCD 计数器的设计使用了 3 个计数为 10（即模 10）计数器的级联结构，分别表示计数 0.1s，1s 和 10s。计数 10 的计数器有一个使能信号，当它计数到 9 的时候会产生一个单周期有效的使能。我们可以利用这些信号将这 3 个计数器“钩”在一起。比如，只有当模 5，000，000 的计数器使能有效时，且 0.1s 和 1s 的计数器计数到 9 时，10s 计数器才能开始计数。代码如示例 4.17 所示。

示例 4.17　码表的串联表示

```
module stop_watch_cascade
   (
   input wire clk,
   input wire go, clr,
   output wire [3:0] d2, d1, d0
   );
   // 声明
   localparam   DVSR = 5000000;
   reg [22:0] ms_reg;
   wire [22:0] ms_next;
   reg [3:0] d2_reg, d1_reg, d0_reg;
   wire [3:0] d2_next, d1_next, d0_next;
   wire d1_en, d2_en, d0_en;
   wire ms_tick, d0_tick, d1_tick;
   // 实体
```

```
// 寄存器
always @ (posedge clk)
begin
    ms_reg <= ms_next;
    d2_reg <= d2_next;
    d1_reg <= d1_next;
    d0_reg <= d0_next;
end
// 次态逻辑
//0.1s 计时器产生单元:模5,000,000
assign ms_next = (clr|| (ms_reg == DVSR && go)) ? 4'b0:
                 (go) ? ms_reg + 1:
                        ms_reg;
assign ms_tick = (ms_reg == DVSR) ? 1'b1:1'b0;
//0.1s 计数器
assign d0_en = ms_tick;
assign d0_next = (clr||(d0_en && d0_reg ==9)) ? 4'b0:
                 (d0_en) ? d0_reg + 1:
                           d0_reg;
assign d0_tick = (d0_reg ==9) ? 1'b1:1'b0;
//1s 计数器
assign d1_en = ms_tick & d0_tick;
assign d1_next = (clr||(d1_en && d0_reg ==9)) ? 4'b0:
                 (d1_en)? d1_reg + 1:
                          d1_reg;
assign d1_tick = (d1_reg ==9)? 1'b1:1'b0;
//10s 计数器
assign d2_en = ms_tick & d0_tick & d1_tick;
assign d2_next = (clr||(d2_en && d2_reg ==9)) ? 4'b0:
                 (d2_en) ? d2_reg + 1:
                           d2_reg;
// 输出逻辑
assign d0 = d0_reg;
assign d1 = d1_reg;
assign d2 = d2_reg;
```

```
endmodule
```

注意，所有的寄存器都是由同一个时钟信号来控制的。本例展示了如何使用单周期的使能信号来保持电路的同步性。另一种较为逊色的方法是使用低一级计数器的输出作为下一阶段的时钟信号。尽管这种方法看起来更简单些，但却违反了同步设计原则，因此是一个非常不好的方法。

设计Ⅱ另外一种描述 3 位数 BCD 计数器的方法是在嵌套的 if 语句中描述整个结构。代码如示例 4. 18 所示。

示例 4. 18　在 IF 语句中描述码表

```
module stop_watch_if
    (
    input wire clk,
    input wire go, clr,
    output wire [3:0] d2, d1, d0
    );
    // 声明
    localparam DVSR = 5000000;
    reg [22:0] ms_reg;
    wire [22:0] ms_next;
    reg [3:0] d2_reg, d1_reg, d0_reg;
    reg [3:0] d2_next, d1_next, d0_next;
    wire ms_tick;
    // 实体
    // 寄存器
    always @ (posedge clk)
    begin
        ms_reg <= ms_next;
        d2_reg <= d2_next;
        d1_reg <= d1_next;
        d0_reg <= d0_next;
    end
    // 次态逻辑
    //0.1s 计时器产生单元:模5,000,000
    assign ms_next = (clr||(ms_reg == DVSR && go)) ? 4'b0:
```

```
                        (go) ? ms_reg + 1:
                               ms_reg;
assign ms_tick = (ms_reg == DVSR) ? 1'b1 : 1'b0;
//3 位BCD 计数器
always @ *
begin
    // 默认值:保持之前的值
    d0_next = d0_reg;
    d1_next = d1_reg;
    d2_next = d2_reg;
    if (clr)
        begin
            d0_next = 4'b0;
            d1_next = 4'b0;
            d2_next = 4'b0;
        end
    else if (ms_tick)
        if (d0_reg! = 9)
            d0_next = d0_reg + 1;
        else                    // 达到XX9
            begin
                d0_next = 4'b0;
            if (d1_reg ! = 9)
                d1_next = d1_reg + 1;
            else                // 达到X99
                begin
                    d1_next = 4'b0;
                    if(d2_reg !  = 9)
                        d2_next = d2_reg + 1;
                    else    // 达到999
                        d2_next = 4'b0;
                    end
                end
end
// 输出逻辑
```

```
    assign d0 = d0_reg;
    assign d1 = d1_reg;
    assign d2 = d2_reg;
endmodule
```

验证电路 为了验证码表的运行，我们可以把前面讲的十六进制 LED 分时复用电路合并，用于显示码表的输出。代码如示例 4.19 所示。注意，LED 的第一个数字被置为数字 0，go 和 clr 信号是由 S3 板上的两个按钮来控制的。

示例 4.19 码表的测试电路

```
module stop_watch_test
   (
    input wire clk,
    input wire [1:0] btn,
    output wire [3:0] an,
    output wire [7:0] sseg
   );
   // 信号的声明
   wire [3:0] d2, d1, d0;
   // 例化7 段数码管模块
   disp_hex_mux disp_unit
      (.clk(clk),.reset(1'b0),
       .hex3(4'b0),.hex2(d2),.hex1(d1),.hex0(d0),
       .dp_in(4'b1101),.an(an),.sseg(sseg));
   // 例化码表
   stop_watch_if counter_unit
      (.clk(clk),.go(btn[1]),.clr(btn[0]),
       .d2(d2),.d1(d1),.d0(d0) );
endmodule
```

4.5.3 FIFO 缓冲器

FIFO（First-In-First-Out）缓冲器是两个子系统之间的“弹性”存储器，结构框图如图 4-10 所示。FIFO 有两个控制信号：wr 和 rd，分别用于写操作和读操作。当 wr 有效时，输入数据被写入缓冲器中。读操作要稍难理解一些。通常来

说，FIFO 缓冲器的“头部”一直都是可访问的，因此在任何时候都能对 FIFO 进行读操作。读信号 rd 扮演着“移除”的角色。当 rd 有效时，第一个进入 FIFO 的数据（即头部）被移除，第二个数据马上变为可访问项。

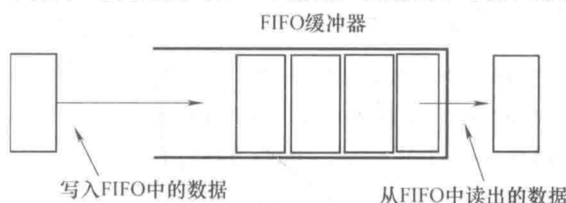


图 4-10 FIFO 缓冲器结构图

FIFO 缓冲器是许多应用中的关键器件，经过优化的实现可能会异常复杂。在这一小节中，我们介绍一个简单、实用的基于循环队列的设计。更加高效，针对特定器件的实现可以查阅 Xilinx 文档。

基于循环队列的实现 构造 FIFO 缓冲器的其中一种方法是在寄存器组上添加一个控制电路。寄存器文件中寄存器被排列成循环队列，并使用两个指针访问。写指针指向队列的头部，而读指针指向队列的尾部，指针根据每次读或者写操作向前推进。8 个 word 的循环队列的操作如图 4-11 所示。

FIFO 缓冲器通常包含两个状态信号，full 和 empty，分别表示 FIFO 满（即不能被写入）和空（即不能被读出）。当读指针等于写指针时，这两个状态的其中一种就会发生，如图 4-11a、f 以及 i 所示。控制器最难设计的地方是获取一个区分这两种状态的机制。一种机制是使用两个触发器去跟踪 empty 和 full 状态，在系统初始化时，这两个触发器分别置位为 1 和 0，然后在每个时钟周期根据 wr 信号和 rd 信号值来修改这两个触发器的值。代码如示例 4. 20 所示。

示例 4. 20 FIFO 缓冲器

```
module fifo
   #(
   parameter B =8, // 每个字的比特数
             W =4  // 地址比特数
   )
   (
   input wire clk, reset,
   input wire rd, wr,
   input wire [B-1:0] w_data,
   output wire empty, full,
```

```
output wire [B-1:0] r_data
);
//信号声明
reg [B-1:0] array_reg [2**W-1:0];  //寄存器数组
reg [W-1:0] w_ptr_reg, w_ptr_next, w_ptr_succ;
reg [W-1:0] r_ptr_reg, r_ptr_next, r_ptr_succ;
reg full_reg, empty_reg, full_next, empty_next;
wire wr_en;
//实体
//寄存器写操作
always @(posedge clk)
    if (wr_en)
        array_reg[w_ptr_reg] <= w_data;
//寄存器读操作
assign r_data = array_reg[r_ptr_reg];
//只有当FIFO 未满时,写使能有效
assign wr_en = wr & ~full_reg;
//FIFO 控制逻辑
//读写指针寄存器
always @(posedge clk, posedge reset)
    if (reset)
        begin
            w_ptr_reg <= 0;
            r_ptr_reg <= 0;
            full_reg <= 1'b0;
            empty_reg <= 1'b1;
        end
    else
        begin
            w_ptr_reg <= w_ptr_next;
            r_ptr_reg <= r_ptr_next;
            full_reg <= full_next;
            empty_reg <= empty_next;
        end
//读写指针的次态逻辑
```

```
always @ *
begin
    // 递增指针值
    w_ptr_succ = w_ptr_reg + 1;
    r_ptr_succ = r_ptr_reg + 1;
    // 默认值:保持之前的值
    w_ptr_next = w_ptr_reg;
    r_ptr_next = r_ptr_reg;
    full_next = full_reg;
    empty_next = empty_reg;
    case ({wr, rd})
        //2'b00 :无操作
        2'b01:// 读
            if (~empty_reg) // 未空
                begin
                    r_ptr_next = r_ptr_succ;
                    full_next = 1'b0;
                    if (r_ptr_succ == w_ptr_reg)
                        empty_next = 1'b1;
                end
        2'b10:// 写
            if (~full_reg) // 未满
                begin
                    w_ptr_next = w_ptr_succ;
                    empty_next = 1'b0;
                    if (w_ptr_succ == r_ptr_reg)
                        full_next = 1'b1;
                end
        2'b11: // 写和读
            begin
                w_ptr_next = w_ptr_succ;
                r_ptr_next = r_ptr_succ;
            end
    endcase
end
```

```
    // 输出
    assign full = full_reg;
    assign empty = empty_reg;
endmodule
```

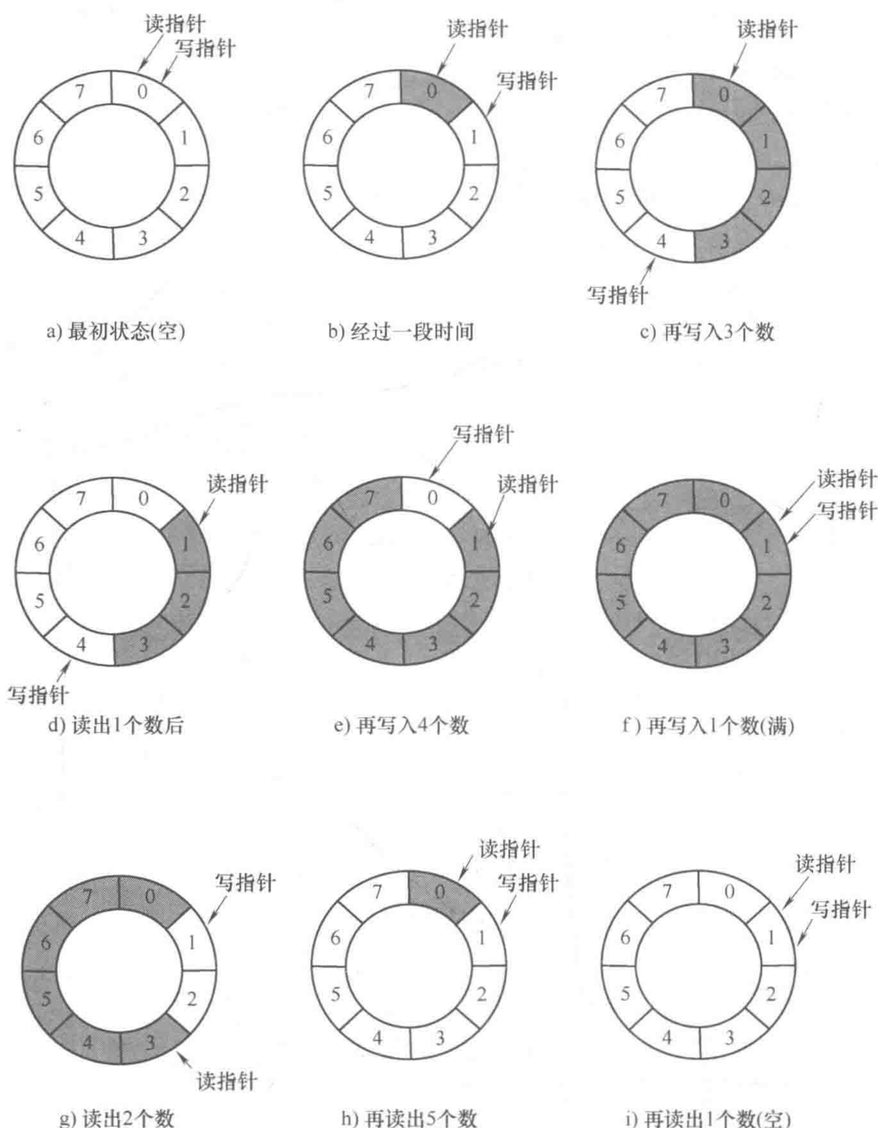


图 4-11　基于循环队列的 FIFO 缓冲器

这段代码分为寄存器组和 FIFO 控制器两个部分。控制器包含了两个指针以及两个状态触发器。次态逻辑检测 wr 和 rd 信号并采取相应的动作。例如，分析

下 case 语句中“10”的情况，它表明只发生了写操作。首先检查状态触发器是否处于非满状态，如条件满足，便将写指针向前推进一个地址，并清零空状态触发器。如果再存储一个数据可能会让 FIFO 处于满状态，则这一情况会发生在新的写指针“追上”读指针时，用 w_ptr_succ == r_ptr_reg 表达式表示。

验证电路　验证电路用于检查 $2^4 \times 3$ FIFO 缓冲器的运行。我们使用 3 个拨动开关去生成输入数据并使用两个按钮产生 wr 信号以及 rd 信号。3bit 的 readout 和 full、empty 状态信号用 5 个分立的 LED 灯显示。由于机械接触具有抖动干扰，需要用一个去抖电路产生一个干净的单时钟周期信号。去抖模块 debounce 已经在 6.2.1 节中做了详细介绍，但是现在我们可以将其看作是一个已经设计好的模块。原始的按钮输入信号为 btn[0]和 btn[1]，去抖后的信号为 db_btn[0]和 db_btn[1]。代码如示例 4.21 所示。

示例 4.21　FIFO 缓冲器的验证电路

```
module fifo_test
    (
    input wire clk, reset,
    input wire [1:0] btn,
    input wire [2:0] sw,
    output wire [7:0] led
    );
    // 信号声明
    wire [1:0] db_btn;
    // btn [0 ]的去抖电路
    debounce btn_db_unit0
        (.clk(clk),.reset(reset),.sw(btn[0]),
         .db_level(),.db_tick(db_btn[0]));
    // btn [1 ]的去抖电路
    debounce btn_db_unit1
        (.clk(clk),.reset(reset),.sw(btn[1]),
         .db_level(),.db_tick(db_btn[1]));
    // 例化2^2 ×3 的FIFO
    fifo #(.B(3), .W(2)) fifo_unit
        (.clk(clk),.reset(reset),
         .rd(db_btn[0]),.wr(db_btn[1]),.w_data(sw),
         .r_data(led[2:0]),.full(led[7]),.empty(led[6]));
```

```
    // 禁用未使用的LED4J
    assign led[5:3] = 3'b000;
endmodule
```

4.6 文献备注

本章的参考文献信息与第 3 章大致相同。

4.7 实验

4.7.1 可编程的方波生成器

可编程的方波生成器是一个可以按一定的间隔产生变量开（逻辑 1）和关（如逻辑 0）的方波的电路。间隔的持续时间由两个 4bit 无符号整型变量 m 和 n 来控制的。开和关的间隔分别保持 m * 100 ns 和 n * 100 ns（S3 板的晶振周期为 20ns）。设计一个可编程的方波生成器电路。这个电路应该是完全同步的。需要一个逻辑分析器或示波器来验证其工作。

4.7.2 PWM 和 LED 调节器

方波信号的占空比被定义为开间隔（即逻辑 1）在周期中所占的比例。PWM（Pulse Width Modulation，脉宽调制）电路可以生成不同占空比的输出。对于一个 4bit 分辨率的 PWM，4bit 的控制信号 w 用于指定占空比。信号 w 被当做无符号整型数，占空比为 w/16。

1）设计一个 4bit 分辨率的 PWM 电路，利用逻辑分析器或者示波器验证其运行情况；

2）修改 LED 分时复用电路，为一个信号添加 PWM 电路，PWM 电路指定了 LED 数码管显示的时间百分比，可通过改变占空比来控制感知的亮度，通过在逻辑分析器或示波器中观察 1 位 an 信号来验证这个电路；

3）使用新的设计来代替示例 4.19 中的 LED 分时复用电路，并使用 8 位开关的低 4 位来控制占空比，验证电路的正确性，可能需要在一个黑暗的环境中观察调节的效果。

4.7.3 旋转的方形图案电路

在七段数码显示管中，我们可以使能 a、b、f、g 段或者 c、d、e、g 段来构

造一个方形图案。我们要设计这样一个电路：让方形图案在 4 位七段数码管中转动。顺时针循环的图案如图 4-12 所示。电路应该有输入信号 en，用于使能或者暂停转动，以及一个能指定方向（顺时针方向或者逆时针方向）的输入信号 cw。

设计这个电路并在开发板上验证。确保循环的频率应足够慢以便视觉上能够察觉。

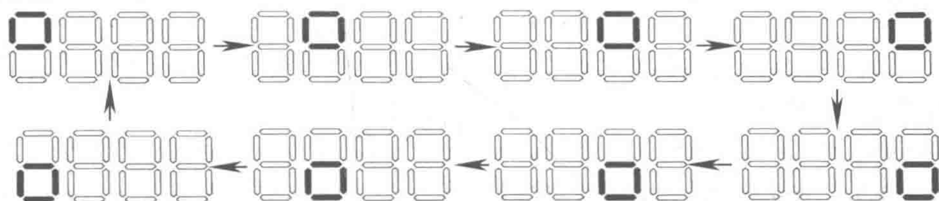


图 4-12　实验 4.7.3 的显示样式

4.7.4　心跳电路

我们想要为开发板创建一个“心跳”。如图 4-13 所示，这个电路只需要以 72Hz 的频率将简单的图形在 4 个七段数码管上重复显示。在开发板上设计这个电路并验证其运行情况。

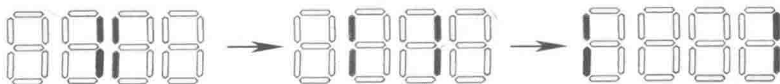


图 4-13　实验 4.7.4 的显示样式

4.7.5　可轮换的 LED 标语电路

原板上有 4 个七段数码显示管，因此在同一时刻只能显示 4 个不同的符号。如果数据是可以不断轮换和移动的，那么就可以显示更多的信息。例如，假设有 10 位数（即“0123456789”），数码显示管可以按“0123”，“1234”，“2345”，……“6789”，“7890”，……“0123”这样的方式显示。电路应有一个输入信号 en，用于使能或暂停循环，以及一个控制轮换方向（向左轮换或者向右轮换）的输入信号 dir。

设计这个电路并在开发板上验证。确保轮换的速率应足够慢以便视觉上能够察觉。

4.7.6　增强的码表

对码表进行如下扩展：

1）增加额外的 up 信号来控制计数的方向，当信号 up 有效时向上计数，否则向下计数；

2）增加分钟数的显示，在 LED 数码管的显示格式应如 M. SS. D，其中 D 表示的是 0.1s，其范围为 0 ~ 9，SS 表示的是秒，其范围是 00 ~ 59，M 表示的是分钟，其范围是 0 ~ 9。

设计这个新的码表并使用测试电路验证其运行情况。

4.7.7 栈

栈是一个后进先出的缓冲器，即后存储的数据先被取出来。通过“push”操作来存储数据至栈中，通过“pop”操作去获得数据，栈的 I/O 引脚和 FIFO 缓冲器的类似，但一般我们使用“push”和“pop”信号来取代 FIFO 中的 wr 和 rd 信号。使用寄存器组设计一个栈并使用测试电路来验证运行情况，可以参考示例 4.21。

第 5 章　有限状态机

5.1　引言

有限状态机（FSM）用来对具有有限个内部转换状态的系统进行建模。这些状态的转换依赖于当前的状态和外部输入。与常规的时序电路不同，有限状态机的状态转换并不是简单的重复模式。有限状态机的次态通常是不固定的，这一点与常规时序电路不同，常规时序电路的次态大多为结构化的元件，如加法器或移位器等。

本章对状态机的基本特点和表示方法进行了概述，并论述 HDL 代码实现方式。在实际应用中，状态机主要用作大型数字系统的一个控制器，检查外部指令和状态，并激活一些控制信号，以控制由常规时序元件组成的数据路径的运行。这类状态机通常又称之为 FSMD（带有数据路径的有限状态机），会在第 6 章进行讨论。

5.1.1　Mealy 输出和 Moore 输出

状态机的基本框图与常规时序电路相同，如图 5-1 所示，一般包含状态寄存器、次态逻辑、输出逻辑。如果一个有限状态机的输出只与其当前状态相关，则该状态机被称为 Moore 状态机；如果一个有限状态的输出既与当前状态有关又与外部输入有关，则该状态机被称为 Mealy 状态机。一个复杂的状态机中可以同时包含 Mealy 输出和 Moore 输出。Mealy 输出和 Moore 输出相似，但不完全一致。理解它们细微差别的关键在控制器的设计上。第 5.3.1 节的例子说明了这两种类型输出的行为和结构特点。

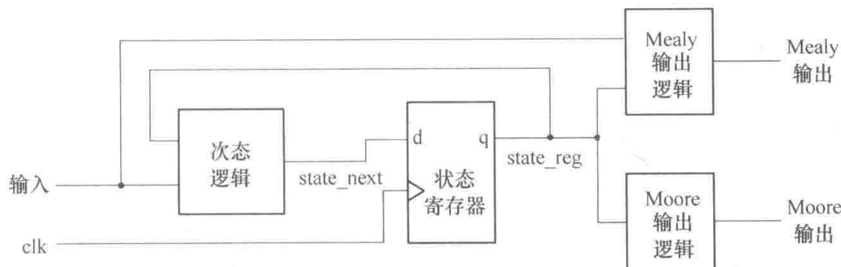


图 5-1　同步状态机框图

5.1.2　有限状态机表示方法

状态机通常由一个抽象的状态图或 ASM 图（算法状态机图）详细说明，对状态机的输入、输出、状态和转移进行图形化表示。两种表示方法提供了同样的信息。状态图表示方法比较适合简单的应用，而 ASM 图表示法更像是一个流程图，更适合描述具有复杂转移条件和行为的应用。

状态图　状态图由若干节点组成，节点表示状态（由圆圈表示），并通过条件转移弧线进行标注。一个单节点和条件转移弧线如图 5-2a 所示。逻辑表达式由相关输入信号和与之相关的转移弧线表示，表示特定的条件。当条件表达式计

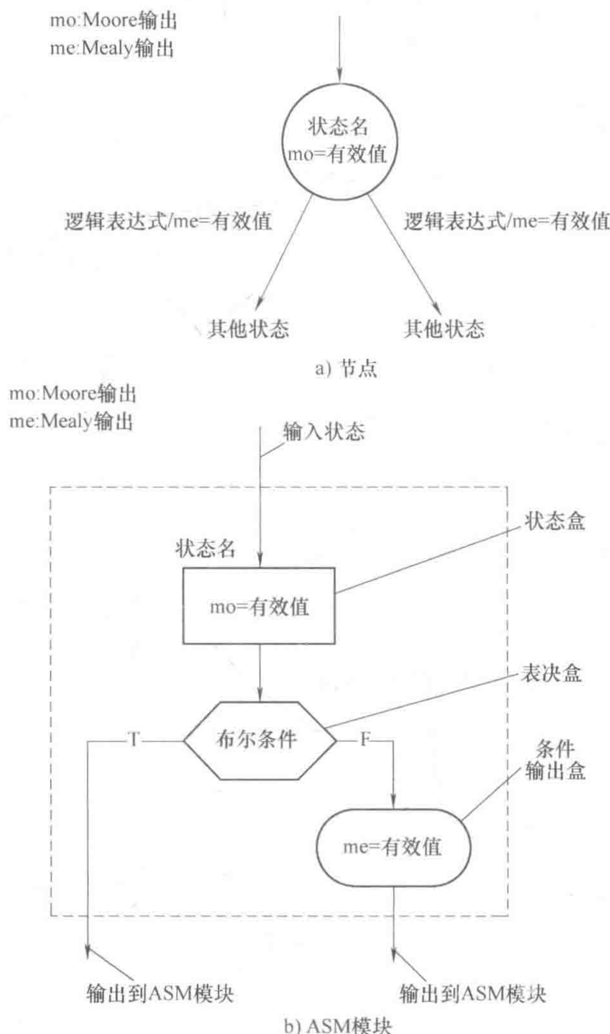


图 5-2　状态符号

算值为真时，转移生效。

由于 Moore 输出值只与当前状态相关，因此输出值放在圆圈内。而 Mealy 输出值与当前状态和外部输入都相关，因此与转移弧线相关。为了减少状态图的臃肿，只有置位的输出值被列出。其他（未置位）的输出值取默认值。

图 5-3a 所示状态图描述的状态机有 3 个状态，2 个外部输入信号（a 和 b），一个 Moore 输出信号（y1）和一个 Mealy 输出信号（y0）。当状态机处于 s0 状态或 s1 状态时，y1 置为有效。当 FSM 处于 s0 状态，且 a、b 信号的值为“11”时，y0 置为有效。

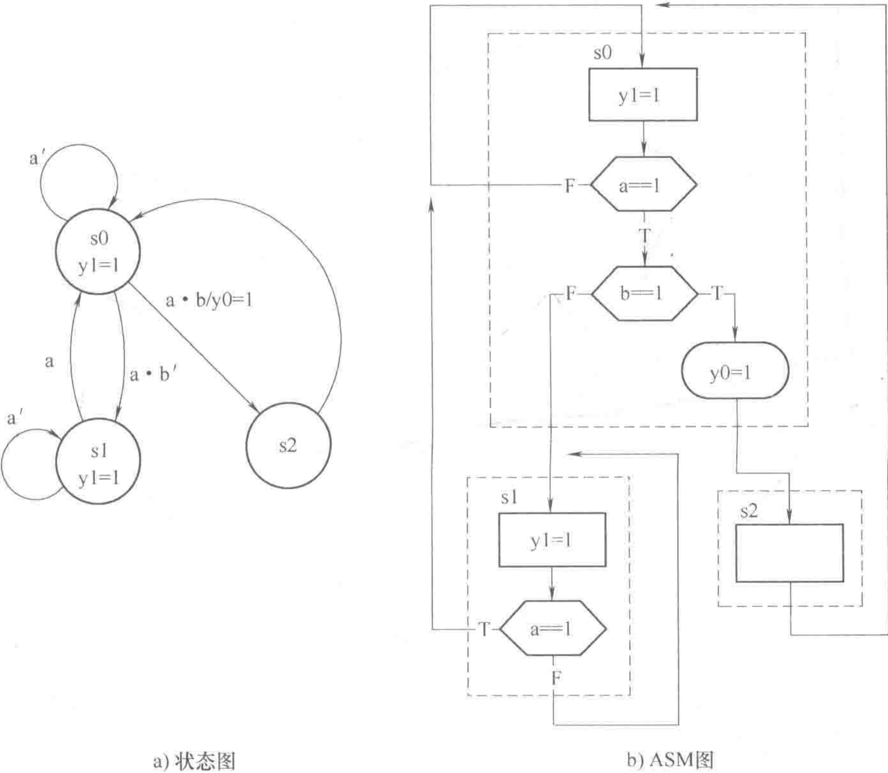


图 5-3 状态机示例

ASM 图 ASM 图是由 ASM 块组成的网络。每一个 ASM 模块包含 1 个状态框和一个可选的表决框及条件输出框。图 5-2b 为一个典型的 ASM 块。

状态框用来表示状态机的一个状态，同时在状态框内明确了 Moore 输出值。状态框只有一个输出路径。表决框测试输入条件并决定走哪条路径。表决框有两

个出口，标记为 T 和 F，分别表示条件为真和条件为假。条件限定框列出了置位的 Mealy 输出值，并通常放置在表决框的后面。它表明只有当表决框中相应的条件满足时，列出的输出信号才将被激活。

状态图可以很容易转换为 ASM 图，反之亦然。图 5-3b 给出了由状态图转换后的 ASM 图。

5.2 状态机编码设计

状态机编码设计与常规时序电路编码设计相似。首先分离状态寄存器，然后使用组合逻辑对次态逻辑和输出逻辑进行编码。主要的差异体现在次态逻辑。对于有限状态机而言，其次态逻辑的代码需要按照状态图或 ASM 图中的流程进行设计。

出于清晰和灵活性的考虑，我们使用符号常量表示状态机的状态。例如，图 5-3 中的状态可以采用下面的方式进行定义：

```
localparam [1:0] s0 = 2'b00,
                 s1 = 2'b01,
                 s2 = 2'b10;
```

综合时，软件通常能识别状态机结构，并将这些符号常量映射为不同的二进制表示法（例如，one-hot 编码），这一过程称之为状态分配。

完整的状态机编码如示例 5.1 所示，包含状态寄存器、次态逻辑、Moore 输出逻辑和 Mealy 输出逻辑。

示例 5.1　FSM 举例

```
module fsm_eg_mult_seg
    (
    input wire clk, reset,
    input wire a, b,
    output wire y0, y1
    );
    // 状态标志声明
    localparam [1:0] s0 = 2'b00,
                     s1 = 2'b01,
                     s2 = 2'b10;
    // 信号声明
    reg [1:0] state_reg, state_next;
```

```
    // 状态寄存器
    always @ (posedge clk, posedge reset)
        if (reset)
            state_reg <= s0;
        else
            state_reg <= state_next;
    // 次态逻辑
    always @ *
        case (state_reg)
            s0: if (a)
                    if (b)
                        state_next = s2;
                    else
                        state_next = s1;
                 else
                    state_next = s0;
            s1: if (a)
                    state_next = s0;
                 else
                    state_next = s1;
            s2: state_next = s0;
            default: state_next = s0;
        endcase
    // 摩尔型输出逻辑
    assign y1 = (state_reg == s0) || (state_reg == s1);
    // 米利型输出逻辑
    assign y0 = (state_reg == s0) & a & b;
endmodule
```

上述代码的关键部分是次态逻辑。它使用了 case 语句，将 state_reg 信号作为选择表达式。次态（即 state_next 信号）是由当前状态（即 state_reg）和外部输入决定的。每个状态的代码主要参考图 5-3b 所示的每个 ASM 块内的活动。

状态机编码的另外一种方式是将次态逻辑和输出逻辑合并到一个组合逻辑块中，如示例 5.2 所示。

示例 5.2　使用组合逻辑合并的状态机示例

```
module fsm_eg_2_seg
   (
   input wire clk, reset,
   input wire a, b,
   output reg y0, y1
   );
   // 状态标志声明
   localparam [1:0] s0 = 2'b00,
                    s1 = 2'b01,
                    s2 = 2'b10;
   // 信号声明
   reg [1:0] state_reg, state_next;
   // 状态寄存器
   always @ (posedge clk, posedge reset)
      if (reset)
         state_reg <= s0;
      else
         state_reg <= state_next;
   // 下个状态逻辑和输出逻辑
   always @ *
   begin
      state_next = state_reg; // 默认下个状态:相同
      y1 = 1'b0;              // 默认输出:0
      y0 = 1'b0;              // 默认输出:0
      case (state_reg)
         s0: begin
               y1 = 1'b1;
               if (a)
                  if (b)
                     begin
                        state_next = s2;
                        y0 = 1'b1;
                     end
```

```
                    else
                        state_next = s1;
                end
            s1: begin
                    y1 = 1'b1;
                    if (a)
                        state_next = s0;
                end
            s2: state_next = s0;
            default: state_next = s0;
        endcase
    end
endmodule
```

注意，状态机的默认输出值列在了代码的开始部分。

次态逻辑和输出逻辑的代码要实现与 ASM 图的流程一致。一旦得到了详细的状态图或 ASM 图，状态机转换为 HDL 代码几乎是一个机械的过程。示例 5.1 和 5.2 可以作为这一用途的模板。

Xilinx ISE 工具内包含了一个叫 StateCAD 的程序，允许用户以图形的方式画一个 Xilinx 状态图。StateCAD 会将状态图转换为精确的 HDL 代码。可以使用一个简单的例子确定 StateCAD 生成的代码是否是满足要求，尤其对如输出信号。

5.3　设计举例

5.3.1　上升沿检测器

上升沿检测器在输入信号由 0 变为 1 时，会产生一个时钟周期的指示信号（tick），常用来指示时序变化较慢的输入信号的开始时刻。下面分别通过 Moore 状态机和 Mealy 状态机实现上升沿检查器的电路功能，并比较两者的不同。

基于 Moore 的设计　基于 Moore 的边沿检测器的状态图和 ASM 图如图 5-4 所示。0 状态和 1 状态分别用于指示输入信号值为 0 和 1 的情况。在 0 状态时，当输入信号变为 1 时，表示检测到上升沿，状态机跳转到 edge 状态。在 edge 状态中对输出信号 tick 置为有效。典型的时序图见图 5-5 的中间部分。代码如示例 5.3 所示。

a) 状态图　　　　b) ASM图

图 5-4　基于 Moore 状态机的边沿检测器

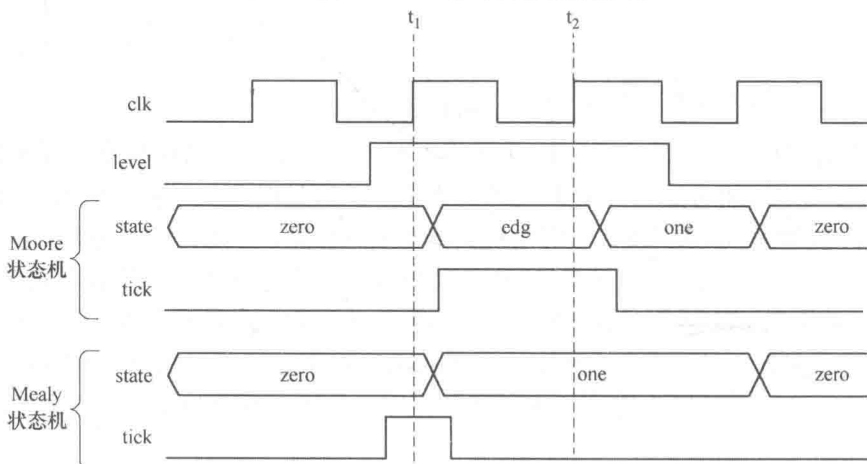


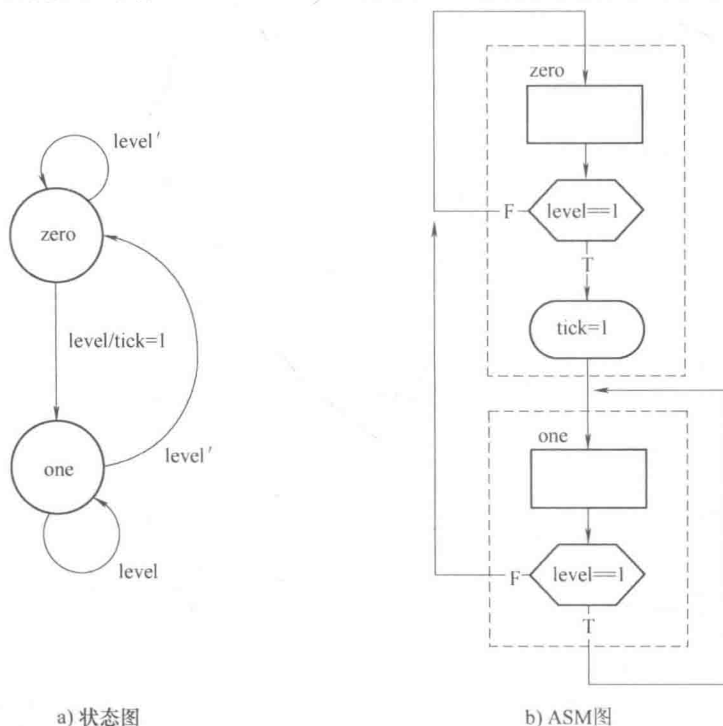
图 5-5　两个边沿检测器的时序图

示例 5.3 基于 Moore 的上升沿检测器

```
module edge_detect_moore
    (
    input wire clk, reset,
    input wire level,
    output reg tick
    );
    // 状态标志声明
    localparam [1:0]
        zero = 2'b00,
        edg = 2'b01,
        one = 2'b10;
    // 信号声明
    reg [1:0] state_reg, state_next;
    // 状态寄存器
    always @ (posedge clk, posedge reset)
        if (reset)
            state_reg <= zero;
        else
            state_reg <= state_next;
    // 下个状态逻辑和输出逻辑
    always @ *
    begin
        state_next = state_reg;  // 默认状态: 相同
        tick = 1'b0;             // 默认输出: 0
        case (state_reg)
            zero:
                if (level)
                state_next = edg;
            edg:
              begin
                tick = 1'b1;
                if (level)
                    state_next = one;
```

```
                else
                    state_next = zero;
            end
        one:
            if ( ~level)
                state_next = zero;
        default: state_next = zero;
    endcase
  end
endmodule
```

基于 Mealy 的设计 基于 Mealy 的边沿检测器的状态图和 ASM 图如图 5-6 所示。0 状态和 1 状态意思相似。当状态机处于 0 状态并且输入信号变为 1 时，输出信号立即置为有效。状态机在时钟信号的上升沿跳转到 1 状态。在 1 状态时，输出信号置为无效。典型的时序图如图 5-5 的底部。注意，由于传输延时，输出信号仍然在下个时钟信号（t_1）的上升沿置位。示例 5.4 给出了状态机的代码实现。



a) 状态图　　b) ASM图

图 5-6 基于 Mealy 的沿检测器

示例5.4　基于 Mealy 的上升沿检测器

```
module edge_detect_mealy
   (
   input wire clk, reset,
   input wire level,
   output reg tick
   );
   // 状态符号声明
   localparam zero = 1'b0,
              one = 1'b1;
   // 信号声明
   reg state_reg, state_next;
   // 状态寄存器
   always @ (posedge clk, posedge reset)
      if (reset)
         state_reg <= zero;
      else
         state_reg <= state_next;
   // 下个状态逻辑和输出逻辑
   always @ *
   begin
      state_next = state_reg;   // 默认状态: 相同
      tick = 1'b0;              // 默认输出: 0
      case (state_reg)
         zero:
            if (level)
               begin
                  tick = 1'b1;
                  state_next = one;
               end
         one:
            if (~level)
               state_next = zero;
         default: state_next = zero;
```

```
        endcase
    end
endmodule
```

直接实现　由于上升沿检测器实现十分简单，因此可以不使用状态机实现。出于比较目的，本书包含其实现部分。如图 5-7 电路图所示，直接实现的上升沿检测器可以解释为只有当前的输入为 1 并且存储在寄存器中之前的输入为 0 时输出被置为有效。相应代码实现如示例 5. 5 所示。

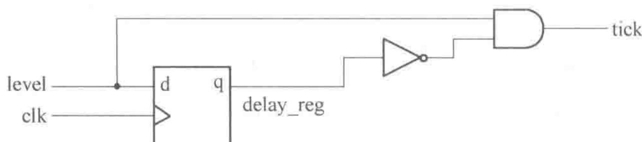


图 5-7　沿检测器的门级实现

示例 5. 5　沿检测器的门级实现

```
module edge_detect_gate
    (
    input wire clk, reset,
    input wire level,
    output wire tick
    );
    // 信号声明
    reg delay_reg;
    // 延时寄存器
    always @ (posedge clk, posedge reset)
        if (reset)
            delay_reg <= 1'b0;
        else
            delay_reg <= level;
    // 解码逻辑
    assign tick = ~delay_reg & level;
endmodule
```

尽管示例 5. 4 和示例 5. 5 中代码的描述看起来完全不同，但它们描述的是同样的电路。如果我们将 0 和 1 分别分配为 0 状态和 1 状态，该电路图便可以通过

状态机得到。

比较　尽管基于 Moore 状态机和 Mealy 状态机的设计都可以在输入信号出现上升沿时给出指示信号（tick），但两者间还是存在微小的差别。基于 Mealy 状态机的设计需要更少的状态，更快的运行速度，但是输出信号位宽差别较大且输入信号的毛刺信号容易传输给输出。

两种设计方式的选择取决于使用输出信号的子系统。在大多数时候，子系统为参考相同时钟信号的同步系统。由于状态机的输出信号只有在时钟信号的上升沿进行采样，只要输出信号在时钟沿附近能保持稳定，位宽和毛刺影响便不大。注意，Mealy 输出信号在 t_1 时刻可以被采样，而 Moore 输出信号在 t_2 时刻才能被采样，因此 Mealy 输出相对于 Moore 输出会快 1 个时钟周期。因此，如果电路应用更注重性能的话，基于 Mealy 设计会更合适一些。

5.3.2　去抖电路

原型板上的滑动开关和按钮为机械装置。对于按钮，按下时，开关会抖动并持续一段时间。抖动导致信号出现如图 5-8 顶部所示的毛刺干扰。抖动通常持续 20ms 左右。去抖电路用来滤波开关切换时产生的毛刺干扰。图 5-8 的底部给出两种基于状态机实现的去抖电路的输出方案。在本节中对第一种设计方式进行讨论，在第 5.5.2 节中对第二种方式进行讨论，在第 6.2.1 节中对另外一种更好的实现方式—基于 FSMD 的设计方式进行讨论。

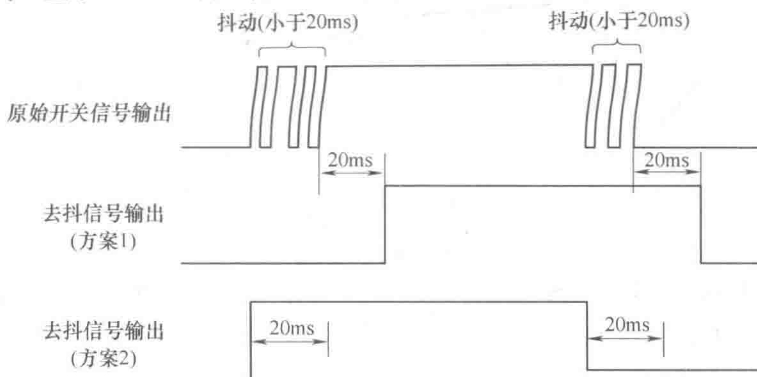


图 5-8　原始的和去抖后的波形

基于状态机的设计要用到一个独立运行的 10ms 计数器和一个状态机。计数器每隔 10ms 产生一个单周期的使能指示信号（m_ tick 信号），状态机通过跟踪该信号信息以确定输入信号是否稳定。在第一种方案中，状态机忽略时间短的抖动，只有当输入信号经过 20ms 的稳定后才会改变去抖后的输出。输出的时序图如图 5-8 中间部分所示。在图 5-9 中给出状态机的状态转移图。其中状态 0 和状

态 1 分别表示开关输出信号 sw 稳定至为 0 或 1。假设初始状态为状态 0，状态机会在信号 sw 变为 1 时迁移到状态 wait1_1。在状态 wait1_1 时，状态机等待 m_tick 变为有效。如果在此状态中，sw 信号变为 0，则表示高电平的持续时间不足以使状态机跳转到状态 0。上述表现都会在状态 wait1_2 和状态 wait1_3 中重复两次。除了 sw 信号持续为 0 外，状态 1 的操作与状态 0 相同。

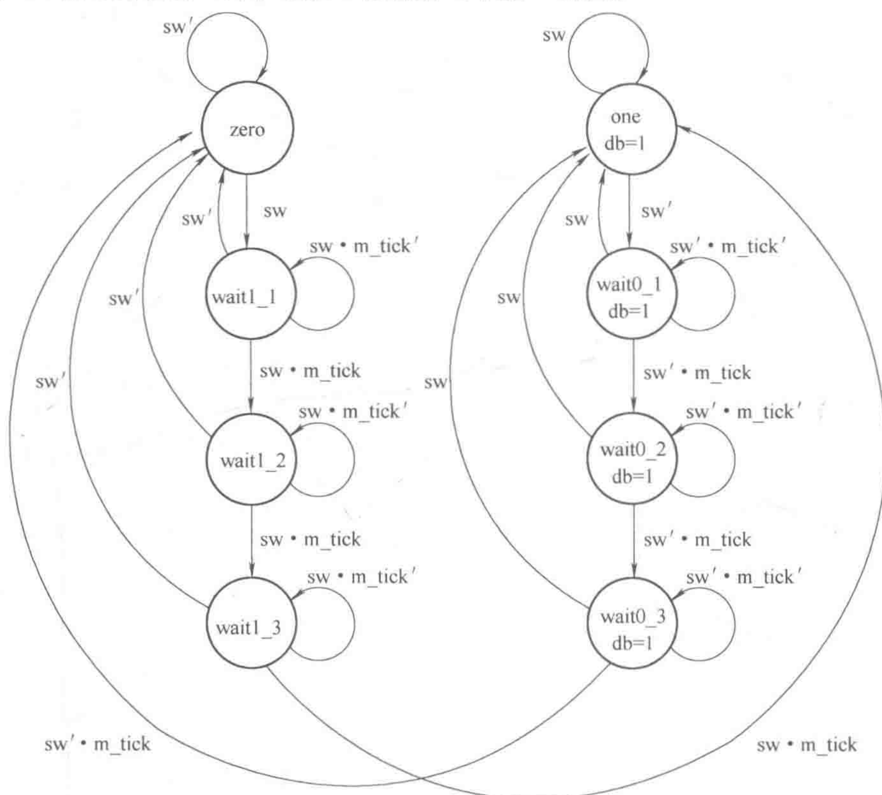


图 5-9　去抖电路的状态转移图

因为 10ms 计数器为独立运行的且 m_tick 信号可能会在任意时刻被置为有效，状态机通过三次检测以保证 sw 信号持续时间至少为 20ms（实际在 20 ~ 30ms 之间）。代码如示例 5.6 所示。里面包含 10ms 计数器和状态机的实现。

示例 5.6　去抖电路状态机实现方式

```
module db_fsm
   (
   input wire clk, reset,
   input wire sw,
   output reg db
```

```
);
// 状态符号声明
localparam  [2:0]
            zero   =3'b000,
            wait1_1 =3'b001,
            wait1_2 =3'b010,
            wait1_3 =3'b011,
            one    =3'b100,
            wait0_1 =3'b101,
            wait0_2 =3'b110,
            wait0_3 =3'b111;
// 计数器bit 数(2^N *20ns =10ms tick )
localparam N =19;
// 信号声明
reg [N-1:0] q_reg;
wire [N-1:0] q_next;
wire m_tick;
reg [2:0] state_reg, state_next;
// 主体
// ==============================================
// 产生10ms 指示信号的计数器
// ==============================================
always @ (posedge clk)
    q_reg <= q_next;
// 次态逻辑
assign q_next = q_reg +1;
// 输出tick
assign m_tick = (q_reg ==0) ? 1'b1:1'b0;
// ==============================================
// 去抖FSM
// ==============================================
// 状态寄存器
always @ (posedge clk, posedge reset)
    if (reset)
        state_reg <= zero;
```

```
    else
        state_reg <= state_next;
// 次态逻辑和输出逻辑
always @ *
begin
    state_next = state_reg;   // 默认状态:相同
    db = 1'b0;                // 默认输出:0
    case (state_reg)
        zero:
            if (sw)
                state_next = wait1_1;
        wait1_1:
            if ( ~ sw)
                state_next = zero;
            else
                if (m_tick)
                    state_next = wait1_2;
        wait1_2:
            if ( ~ sw)
                state_next = zero;
            else
                if (m_tick)
                    state_next = wait1_3;
        wait1_3:
            if ( ~ sw)
                state_next = zero;
            else
                if (m_tick)
                    state_next = one;
        one:
            begin
                db = 1'b1;
                if ( ~ sw)
                    state_next = wait0_1;
            end
```

```
            wait0_1:
                begin
                    db = 1'b1;
                    if (sw)
                        state_next = one;
                    else
                        if (m_tick)
                            state_next = wait0_2;
                end
            wait0_2:
                begin
                    db = 1'b1;
                    if (sw)
                        state_next = one;
                    else
                        if (m_tick)
                            state_next = wait0_3;
                end
            wait0_3:
                begin
                    db = 1'b1;
                    if (sw)
                        state_next = one;
                    else
                        if (m_tick)
                            state_next = zero;
                end
            default: state_next = zero;
        endcase
    end
endmodule
```

5.3.3 测试电路

我们通过如图 5-10 框图所示的抖动计数电路对上升沿检测电路和去抖电路

进行验证。验证电路的输入来自按钮式开关。在图 5-10 的下半部，信号先后经过去抖电路和上升沿检测电路。因此，按钮每按下并被释放一次，就会产生一个单周期的指示信号。指示信号控制使能 8bit 计数器的输入，计数器的值会传输给分时复用的 LED 电路，并显示在开发板上的七段数码管左两位数字上。在图 5-10 的上半部分，输入信号不经过去抖电路而是直接输入到上升沿检测电路，数字会在原型板上的右两位七段式 LED 显示器上进行显示。底部的计数器在按钮弹起时计数一次 0 到 1 状态的跃迁。

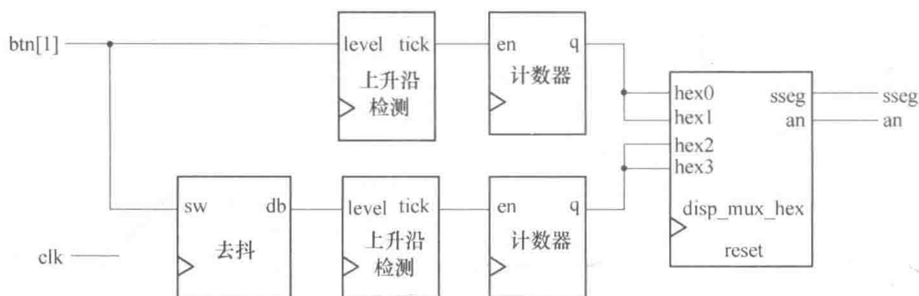


图 5-10　去抖测试电路

示例 5.7 中给出了代码的实现方式。主要是通过元件实例化的方式实现框图的功能。

示例 5.7　去抖电路和上升沿检测电路验证电路实现代码

```
module debounce_test
    (
    input wire clk, reset,
    input wire [1:0] btn,
    output wire [3:0] an,
    output wire [7:0] sseg
    );
    // 信号声明
    reg [7:0] b_reg, d_reg;
    wire [7:0] b_next, d_next;
    reg btn_reg, db_reg;
    wire db_level, db_tick,btn_tick, clr;
    //7 段式LED 多路时间显示模块示例
    disp_hex_mux disp_unit
```

```
(. clk(clk),. reset(reset),
 . hex3(b_reg[7:4]),. hex2(b_reg[3:0]),
 . hex1(d_reg[7:4]), . hex0(d_reg[3:0]),
 . dp_in(4'b1011),. an(an),. sseg(sseg));
// 去抖电路示例
db_fsm db_unit
    (. clk(clk),. reset(reset),. sw(btn[1]),. db(db_level));
// 沿检测电路
always @ (posedge clk)
    begin
        btn_reg <= btn[1];
        db_reg <= db_level;
    end
assign btn_tick = ~btn_reg & btn[1];
assign db_tick = ~db_reg & db_level;
// 双计数器
assign clr = btn[0];
always @ (posedge clk)
    begin
        b_reg <= b_next;
        d_reg <= d_next;
    end
assign b_next = (clr)        ? 8'b0:
                (btn_tick)   ? b_reg + 1:b_reg;
assign d_next = (clr)        ? 8'b0:
                (db_tick)    ? d_reg + 1:d_reg;
endmodule
```

七段数码管显示器显示了累积的由 0 变 1 的反弹边沿个数，以及经过去抖处理的开关输入个数。经过对按钮进行多次的按下和释放操作，我们可以得到一次转换的平均反弹次数。

5.4　文献备注

C. E. Cummings 的文章 “Coding and Scripting Techniques for FSM Designs with

Synthesis-Optimized，Glitch-Free Outputs”对状态机不同的编码风格进行了详细的讨论。

5.5 参考实验

5.5.1 双沿检测器

双沿检测器与上升沿检测器相似，只是双沿检测器要在输入信号由 0 变为 1（上升沿）和由 1 变为 0（下降沿）时输出均需要置一个周期的有效信号。

1）设计一个基于 Moore 状态机的电路，并画出状态图和 ASM 图；

2）通过 ASM 图或状态图得到 HDL 代码；

3）编写测试平台并通过仿真验证代码的运行；

4）将 5.3.3 节中的上升沿检测电路替换成双沿检测电路，并进行验证；

5）用基于 Mealy 状态机的设计重复上述步骤 1～4。

5.5.2 另一种去抖电路

5.3.2 节去抖电路的一个问题是开关转换开始的延时响应。另外一种方法是只对第一个跳变沿进行反应，然后等待一小段时间（至少 20ms）以使输入信号稳定。这种方法的输出时序图如图 5-8 底部。当输入从 0 变为 1 时，FSM 立即响应，然后忽略后面大约 20ms 的输入来避免毛刺。之后，FSM 再开始检查输入的下降沿。按 5.3.2 节中的设计流程设计这种电路。

1）设计电路的状态图和 ASM 图；

2）得到 HDL 代码；

3）基于状态图和 ASM 图得到 HDL 代码；

4）设计测试平台并利用仿真来验证代码的运行情况；

5）使用另一种设计取代 5.3.3 节的去抖电路并验证其运行情况。

5.5.3 停车场占用计数器

考虑一个只有单一的入口和出口的停车场。有两对图像传感器用于监视汽车的活动，如图 5-11 所示。当一个物体在图像发送器和图像接收器之间时，由于光线被遮

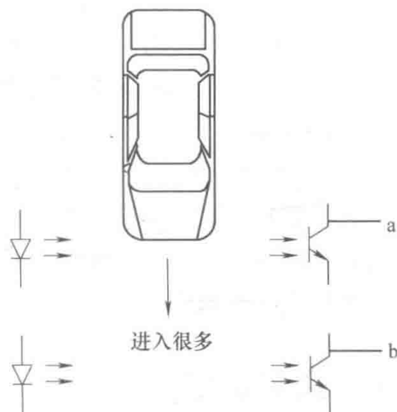


图 5-11　门卫传感器示意图

挡，因此相应的输出就会被置为高电平。通过监视两个传感器的事件，我们可以判断是否有车进出或者某个行人经过。举例来说，如下事件序列表示有一辆车进入了停车场：

- 最开始，两个传感器都处于未遮挡状态（即信号 a 与 b 为“00”）；
- 传感器 a 被遮挡（即信号 a 与 b 为“10”）；
- 两个传感器均被遮挡（即信号 a 与 b 为“11”）；
- 传感器 a 遮挡消失（即信号 a 与 b 为“01”）；
- 两个传感器均遮挡消失（即信号 a 与 b 为“00”）。

设计一个停车场占用计数器步骤如下：

1）设计一个带有两个输入信号（a 和 b），两个输出信号（enter 和 exit）的状态机，当一辆车进入和出来时，分别置位 enter 和 exit 一个周期的有效信号；

2）设计状态机的 HDL 代码；

3）设计一个带有两个控制信号（inc 和 dec）的计数器，用于增减计数，并获得 HDL 代码；

4）将计数器和状态机以及 LED 多路选择电路进行组合，用两个经过去抖的按钮模拟两个传感器的输出功能，验证该停车占用计数器。

第 6 章　带数据路径的有限状态机

6.1　简介

带数据路径的有限状态机（FSMD）是由一个有限状态机和常规时序电路组合而成的。有限状态机有时候被称为控制路径，用来检测外部命令和状态并且生成控制信号，以指定常规时序电路的相应操作。FSMD 通过使用 RT（寄存器传输）方法学的描述来实现系统，使用这种方法时，相应的操作被指定为数据操作和一系列的寄存器传输。

6.1.1　单个 RT 操作

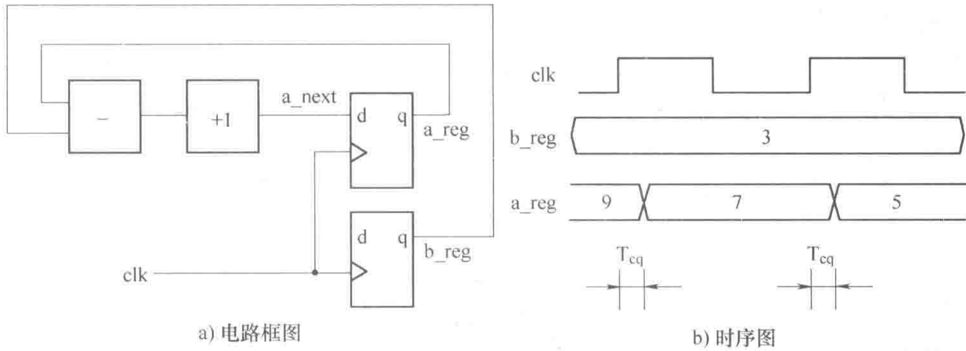
单个 RT 操作指的是对于单个目标寄存器的数据处理和传输。由以下符号表示：

$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, \cdots r_{srcn})$$

r_{dest}是目标寄存器，r_{src1}、r_{src2}及 r_{srcn} 等是源寄存器，而$f(\cdot)$代表当前的操作。上述符号的含义是源寄存器的值传给了由组合逻辑电路实现的函数$f(\cdot)$，而结果传递给目标寄存器的输入，并于下一个时钟沿存入目标寄存器中。以下为几个典型的 RT 操作：

- r1←0. 常量 0 被存入寄存器 r1；
- r1←r1. r1 寄存器的值被重新写回；
- r2←r2 >> 3。寄存器 r2 右移 3 位后写回寄存器 r2；
- r2←r1。将寄存器 r1 的值传输给寄存器 r2；
- i←i + 1。寄存器 i 中的值自加 1 后重新写回到寄存器 i 中；
- d←s1 + s2 + s3。将寄存器 s1、s2、s3 的和写入寄存器 d 中；
- y←a * a。将 a 的二次方写入 y 寄存器中。

单个 RT 操作可以利用表示函数$f(\cdot)$的组合逻辑电路并连接输入和输出寄存器来实现。以 a←a - b + 1 操作为例子，函数$f(\cdot)$会涉及一个减法器和一个加法器，原理框图如图 6-1a 所示。为清晰起见，我们使用“_reg”和“_next”后缀表示寄存器的输入和输出。需要注意的是，单个 RT 操作是由一个内嵌的时钟来同步的。这样，$f(\cdot)$的结果只有在下一个时钟周期才能被存入目标寄存器中。上一次的 RT 操作时序图见图 6-1b。



a) 电路框图

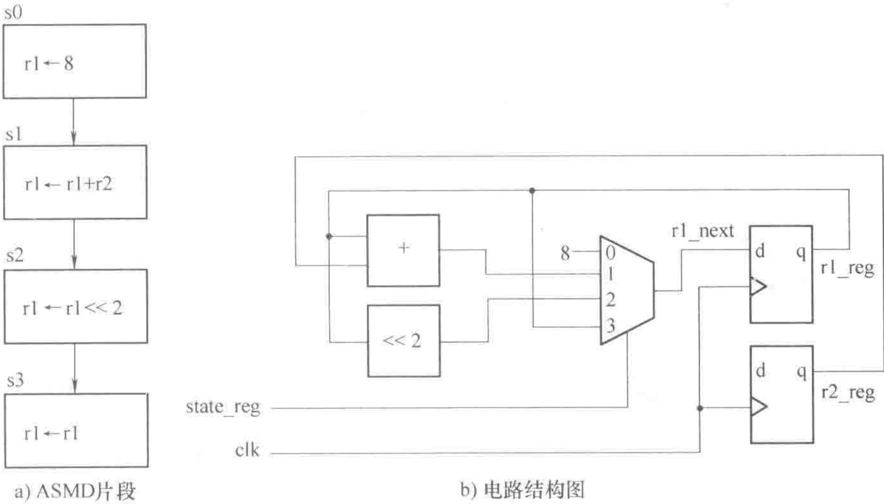
b) 时序图

图 6-1　单个 RT 操作的电路框图和时序图

6.1.2　ASMD 图

基于 RT 方法学电路指定了每一步应该执行的 RT 操作。由于 RT 操作是在一个时钟接着一个时钟的基础上完成的，因此它的时序与状态机的跳转很相似。这样，选择用状态机来指定 RT 算法的顺序就很自然了。我们将 RT 操作纳入到 ASM 图中对 ASM 图进行扩展，将其称之为 ASMD（带数据路径的 ASM）图。这种 RT 操作被当作是另一种类型的活动，可用于使用输出信号的地方。

图 6-2a 为 ASMD 图的一个片段。它包含了将 r1 寄存器初始化为 8，然后再与 r2 寄存器中的值相加，最后将结果左移两位。注意，必须在每个状态都要指定 r1 寄存器。如果 r1 寄存器的值没有变化，则使用如 S3 中所示的 r1←r1 操作来保持当前值。在以后的讨论中，我们假定 r←rRT 操作对于 r 寄存器来说是默



a) ASMD片段

b) 电路结构图

图 6-2　ASMD 片断的实现

认的操作，将不会包含在 ASMD 图中。要实现一个含 RT 操作的 ASMD 图涉及使用一个多路选择电路将下一个期望值发送至目标寄存器中。例如，图 6-2b 所示的电路结构可以通过前述章节中的一个 4 选 1 的多路选择器来实现。状态机的当前状态（即状态寄存器的输出）控制多路选择器的选择信号并对期望的 RT 操作结果进行选择。

RT 操作也可以通过一个条件判断输出结构指定，如图 6-3a 中的寄存器 r2。根据条件 a > b，状态机执行分支 r2←r2 + a 或者分支 r2←r2 + b。注意，在 ASMD 模块内所有操作都是并行执行的。我们需要实现 a > b、r2 + a 以及 r2 + b 操作，并使用多路选择器将期望值发送至 r2。结构框图如图 6-3b 所示

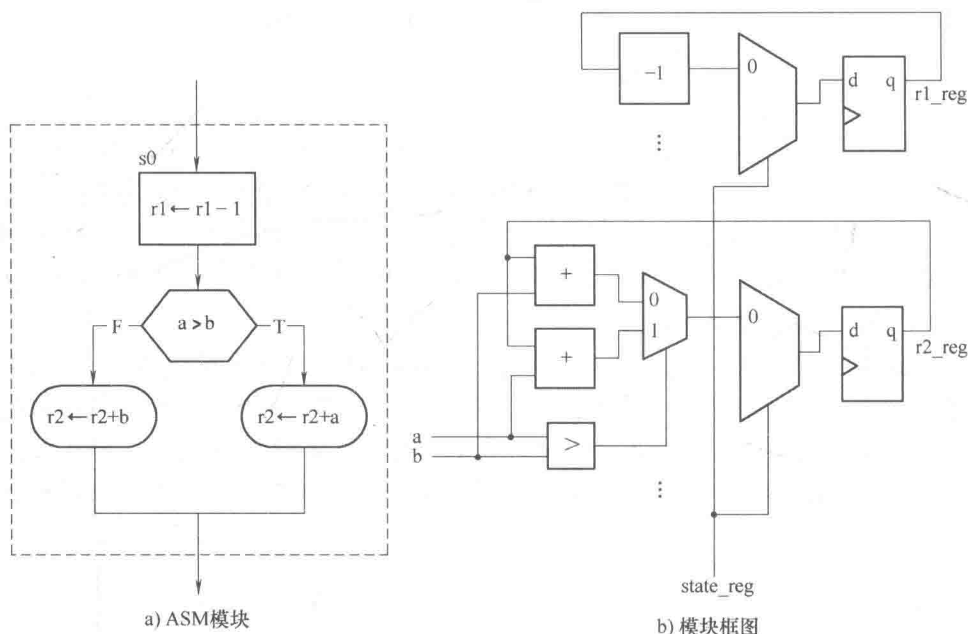


图 6-3　在一个条件输出盒子中实现 RT 操作

6.1.3　带寄存器的判决盒

乍看起来 ASMD 图与正常的程序流程图非常相似。主要的不同之处是在 ASMD 图中，RT 操作是由模块内嵌的时钟信号来控制的，并且目标寄存器的值的更新发生在 FSMD 退出当前 ASMD 块时，而不是在 ASMD 块内。表达式 r←r − 1 实际上意味着：

- r_next = r_reg − 1；
- r_reg <= r_next 在时钟上升沿执行（即当 FSMD 退出当前块的时候）。

当寄存器用于判决盒中时，这种“延时存储”可能会引入微妙的错误。思

考图 6-4a 所示的 FSMD 片段。寄存器 r 在状态盒中递减且在判决盒中使用。由于寄存器 r 会一直等到 FSMD 退出块时才会更新，寄存器 r 中的旧值被用于判决盒比较。如果期望的是 r 的新值，那么应该在判决盒中使用组合逻辑的输出（即 r_next）（即使用 r_next == 0 的表达来取代 r == 0），如图 6-4b 所示。

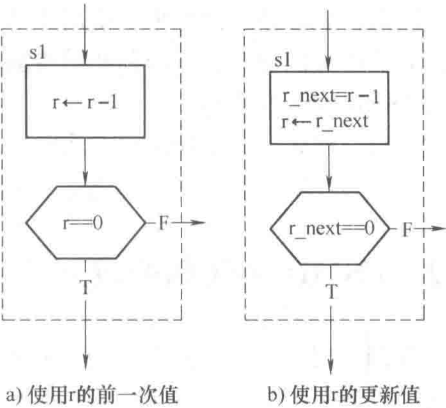


图 6-4　延时存储对 ASM 模块的影响

FSMD 模块框图 FSMD 的模块框图从形式上可以分为数据路径和控制路径，如图 6-5 所示。数据路径的执行需要进行 RT 操作。它包含：

- 数据寄存器：存储中间计算结果；
- 功能单元：由一系列的 RT 操作完成功能；
- 布局网络：在功能单元和存储寄存器之间布局数据通道。

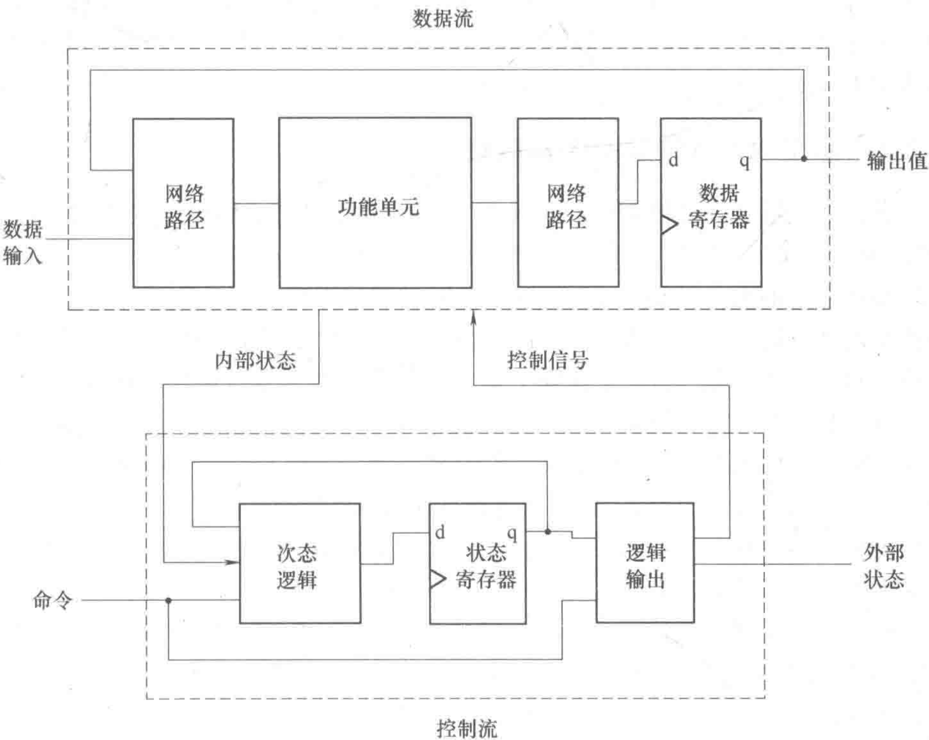


图 6-5　FSMD 框图

数据路径根据控制信号执行期望的 RT 操作并生成相应的内部状态信号。

控制路径是一个有限状态机。它是一个常规的有限状态机，包含状态寄存器、次态逻辑以及输出逻辑。使用外部命令信号和数据路径的状态指示信号作为输入并产生控制信号，以控制数据流的操作。状态机还会生成外部状态信号用于指示 FSMD 的运行状态。

注意，尽管一个 FSMD 包含两种类型的时序电路，但由于两种电路都是受到相同的时钟控制，因此 FSMD 仍是一个同步系统。

6.2 FSMD 的代码开发

我们使用一个改进的去抖电路来展示如何得到 FSMD 的代码。尽管去抖电路在 5.3.2 节中使用了一个状态机和一个计时器（一个常规时序电路），但是并非是基于 RT 方法学实现的，因此这两个功能单元独立运行，且状态机并没有控制计时器。由于每隔 10ms 产生的使能信号可以在任意时刻有效，因此，当计时器在等待状态 1_1 或者 0_1 的时候检测到第一次滴嗒时，状态机并不知道实际已经经历了多长时间。这样，等待周期虽然在 20 ~ 30ms 之间，但并非准确值。这种误差可以通过使用 RT 方法学解决。在本节中，我们使用这个改进的去抖电路来说明 FSMD 的代码开发过程。

6.2.1 基于 RT 方法学的去抖电路

我们在使用状态机控制计时器的初始化以期获得准确的时间间隔时要用到 RT 理论。ASMD 表如图 6-6 所示。这种扩展的电路包括两个输出信号：db_level，循环跳转电路的输出；db_tick，通过一个时钟周期脉冲有效沿来判断从零状态到一状态循环跳转。零状态和一状态分别意味着输入 sw 已经分别稳定到 0 和 1。等待 1 和等待 0 状态一般用于滤除毛刺。sw 信号必须是某一稳定的时间计数值，否则这种状态跳转将会被认为是一个干扰信号。数据流包括一个 21 位宽的寄存器 q。假设 FSMD 最初处于零状态。当 sw 输入信号变为 1 时，FSMD 跳转到 wait1 状态并且初始化 q 值为“1…1”。当电路处于 wait1 状态的时候，q 寄存器在每个时钟周期都会递减一次。如果 sw 信号保持为 1，FSMD 将会反复进入这个状态直到 q 寄存器中的值为“0…0”，然后进入 one 状态。

50MHz（即周期为 20ns）的系统时钟曾经通常被用在原型板上。由于 FSMD 有 2^{21} 个时钟周期处于 wait1 状态，大约是 40ms（i. e.，$2^{21} * 20ns$）。我们可以通过修改 q 寄存器的初始值来获得期望的时间间隔。

有两种方法可以追溯 HDL 编码方式：一种是数据流组成的白盒描述，另一种是数据流组成的黑盒描述。

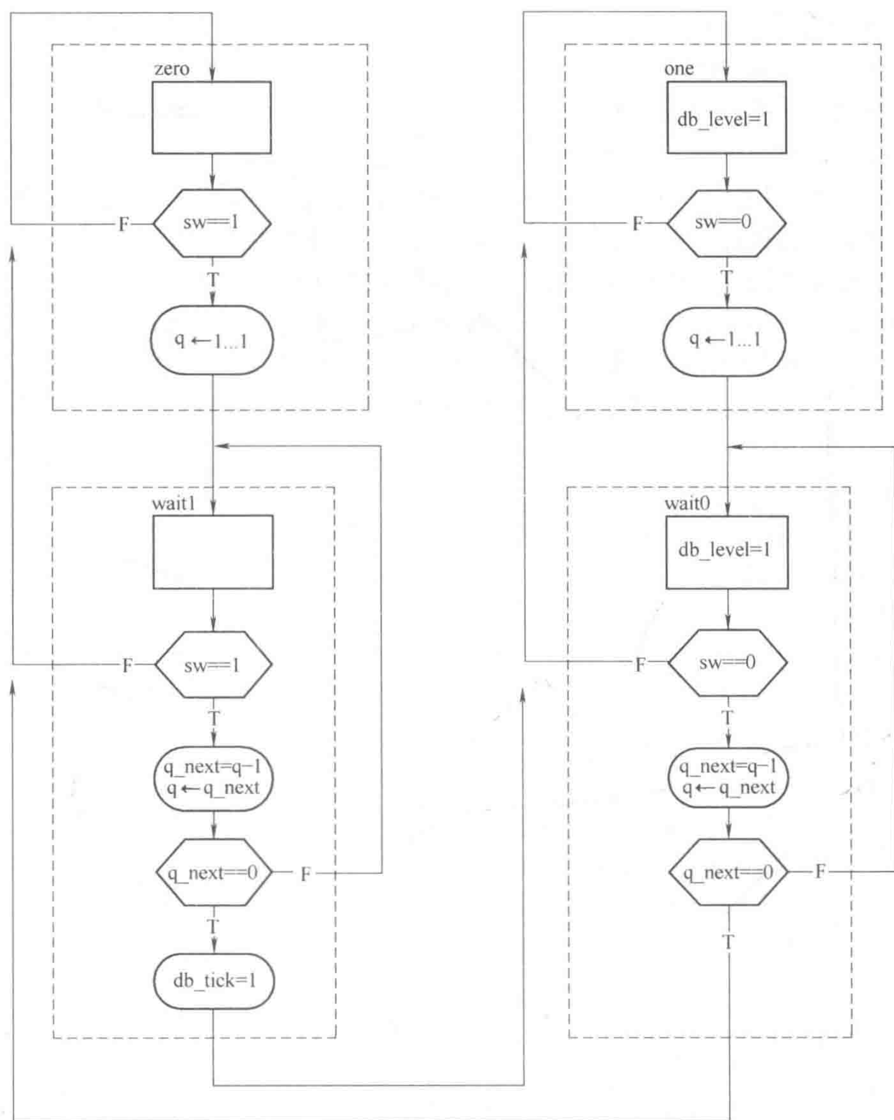


图 6-6　基于循环跳转电路的 ASMD 表

6.2.2　带有数据路径元件的编码

进行 FSMD 代码开发的第一件事就是分离状态机控制流和关键数据流。从 ASMD 图可知，首先要识别数据和相关控制信号中的关键元件，然后使用独立的代码段来描述这些元件。

去抖电路的 ASMD 图的关键数据路径元件是一个位宽为 21 位的递减计数器，它可以：

● 使用特定值初始化；
● 向下计数或暂停；
● 当计数值到 0 时会置一个状态信号有效。

我们可以创建一个二进制计数器，利用 q_load 信号加载初始值以及利用 q_dec 信号使能计数。计数器也会产生一个状态信号 q_zero，当计数到 0 时有效。完整的数据路径是由一个 q 寄存器和定制的递减计数器的次态逻辑组成。还包含一个比较电路用于生成 q_zero 状态信号。控制路径包含一个状态机，它使用了 sw 输入信号和 q_zero 状态信号，并根据 ASMD 图中的期望行为置控制信号 q_load 和 q_dec 有效。HDL 代码遵循了数据路径说明和 ASMD 图，如示例 6.1 所示。

示例 6.1　带有明确数据路径元件的去抖电路

```
module debounce_explicit
  (
    input wire clk, reset,
    input wire sw,
    output reg db_level, db_tick
  );
  // 符号状态声明
  localparam  [1:0]
              zero   =2'b00,
              wait0  =2'b01,
              one    =2'b10,
              wait1  =2'b11;
  // 计数器位宽(2^N *20ns =40ms)
  localparam N =21;
  // 信号定义
  reg[1:0]state_reg, state_next;
  reg[N-1:0]q_reg;
  wire[N-1:0]q_next;
  wire q_zero;
  reg q_load, q_dec;
// 代码部分
//FSMD 状态和数据寄存器
    always @(posedge clk, posedge reset)
      if(reset)
```

```
        begin
          state_reg <= zero;
          q_reg <=0;
        end
    else
        begin
          state_reg <= state_next;
          q_reg <= q_next;
        end
//FSMD 计数器数据路径和下一状态逻辑
assign q_next = (q_load)? {N{1'b1}}:  //load 1..1
             (q_dec)? q_reg - 1:       // 递减
                      q_reg;
// 状态信号
assign q_zero = (q_next ==0);
//FSMD 控制路径的下一状态逻辑
always @*
begin
    state_next = state_reg;  // 默认保持上一个状态
    q_load =1'b0;            // 默认输出为低电平
    q_dec =1'b0;             // 默认输出为低电平
    db_tick =1'b0;           //默认输出为低电平
    case (state_reg)
      zero:
         begin
           db_level =1'b0;
           if(sw)
              begin
                state_next = wait1;
                q_load =1'b1;
              end
         end
      wait1:
         begin
           db_level =1'b0;
```

```
            if( sw)
                begin
                  q_dec = 1'b1;
                  if( q_zero)
                      begin
                        state_next = one;
                        db_tick = 1'b1;
                      end
                end
            else //sw ==0
                state_next = zero;
          end
      one:
          begin
            db_level = 1'b1;
            if( ~sw)
                begin
                  state_next = wait0;
                  q_load = 1'b1;
                end
          end
      wait0:
          begin
            db_level = 1'b1;
            if( ~sw)
                begin
                  q_dec = 1'b1;
                  if( q_zero)
                      state_next = zero;
                end
            else//sw == 1
                state_next = one;
          end
      default: state_next = zero;
  endcase
```

```
  end
endmodule
```

6.2.3 带有隐含数据路径元件的编码

另一种代码风格是在状态机控制路径中嵌入 RT 操作。我们仅仅使用对应的状态机状态来列举 RT 操作，而不是定义数据路径元件。去抖电路的编码如示例 6.2 所示。

示例 6.2　带有隐含数据路径元件的去抖电路

```
module debounce
  (
    input wire clk,reset,
    input wire sw,
    output reg db_level,db_tick
  );
  // 符号化状态声明
  localparam  [1:0]
              zero   =2'b00,
              wait0  =2'b01,
              one    =2'b10,
              wait1  =2'b11;
  // 计数器位宽(2^N *20ns =40ms )
  localparam N =21;
  // 信号声明
  reg[N-1:0]q_reg,q_next;
  reg[1:0]state_reg,state_next;
  // 代码部分
  //FSMD 状态和数据寄存器
   always @ (posedge clk, posedge reset)
     if(reset)
         begin
           state_reg <= zero;
           q_reg <=0;
         end
```

```
  else
      begin
        state_reg <= state_next;
        q_reg <= q_next;
      end
// 下一状态逻辑和数据流功能单元/布局
always @*
begin
  state_next = state_reg;   // 默认为保持前面状态
  q_next = q_reg;           // 默认q 值保持不变
  db_tick = 1'b0;           // 默认输出:0
  case(state_reg)
    zero:
        begin
          db_level = 1'b0;
          if(sw)
              begin
                state_next = wait1;
                q_next = {N{1'b1}}; //load 1..1
              end
        end
    wait1:
        begin
          db_level = 1'b0;
          if(sw)
              begin
                q_next = q_reg - 1;
                if(q_next == 0)
                    begin
                      state_next = one;
                      db_tick = 1'b1;
                    end
              end
          else// sw == 0
              state_next = zero;
```

```
            end
        one:
            begin
                db_level = 1'b1;
                if( ~sw)
                    begin
                        state_next = wait0;
                        q_next = {N{1'b1}};//load 1..1
                    end
            end
        wait0:
            begin
                db_level = 1'b1;
                if( ~sw)
                    begin
                        q_next = q_reg - 1;
                        if(q_next ==0)
                            state_next = zero;
                    end
                else// sw ==1
                    state_next = one;
            end
        default: state_next = zero;
    endcase
  end
endmodule
```

代码包含一段存储器代码和一段组合逻辑编代码。前者由状态机中的状态寄存器和数据路径中的数据寄存器组成。后者主要由状态机中控制路径的次态逻辑指定。次态数据寄存器中的值是由单独的状态指定的，而不是控制信号。包括功能单元和路由网络数据路径的次态逻辑，也据此创建。

6.2.4　对比

带有隐含数据路径元件的代码编码本质遵循的是 ASMD 图。我们仅仅是将 ASMD 图转换为 HDL 语言描述。尽管这种方法更简单而且更容易描述，我们要

依赖于对综合软件对数据路径进行构造，可控制的地方很少。这点可以通过一个例子做最好的诠释。思考图 6-7 所示的 ASMD 片段，隐含的描述变成了：

```
case(state_reg)
    s1:
    begin
        d1_next = a * b;
        ...
    end
    s2:
    begin
        d2_next = b * c;
    end
    s3:
    begin
        d3_next = a * c;
        ...
    end
        ...
endcase
```

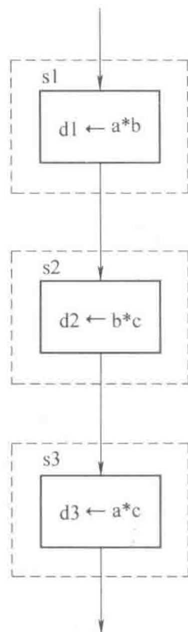


图 6-7　通过共享的 ASMD 代码片段

通过综合软件可能推断出 3 个乘法器。由于组合逻辑的乘法器是一个复杂的电路，对电路进行共享更加高效。我们可以使用显示描述来分离乘法器：

```
case(state_reg)
    s1:
    begin
        in1 = a;
        in2 = b;
        d1_next = m_out;
        ...
    end
    s2:
        begin
        in1 = b;
        in2 = c;
        d2_next = m_out;
```

```
        ...
        end
    s3:
    begin
        in1 = a;
        in2 = c;
        d3_next = m_out;
    ...
    end
    ...
endcase
...
// 单个乘法器的详细描述
//always 模块的外部
assign m_out = in1 * in2;
```

这种编码方式保证了在综合的时候仅有一个乘法器被推断出来。这种隐式描述和显示描述可以混合在一个复杂 FSMD 设计中。出于代码简洁高效考虑，我们经常分离和提取复杂的数据流元件。

6.2.5 测试电路

5.3.3 节讨论的去抖测试电路可用于验证新设计的运行情况。由于被修改的去抖电路的输出包含一个单时钟周期的标识信号，因此去抖电路后面不需要再加边沿检测器。修改后的模块结构框图如图 6-8 所示，相应代码如示例 6.3 所示。

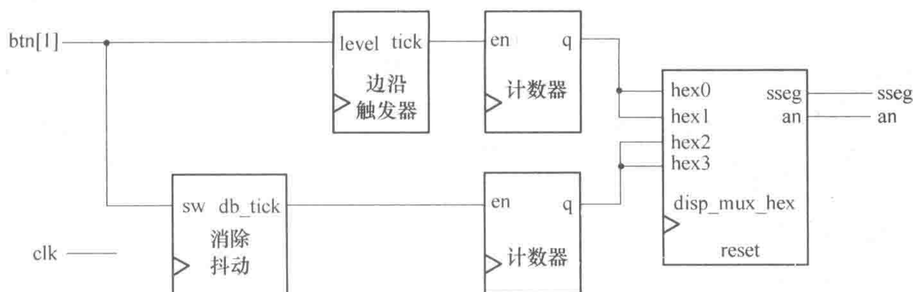


图 6-8　跳转电路测试电路

示例 6.3　去抖电路的验证电路

```
module debounce_fsmd_test
```

```
(
input wire clk,reset,
input wire[1:0]btn,
output wire[3:0]an,
output wire[7:0]sseg
);
// 信号声明
reg  [7:0]  b_reg,d_reg;
wire  [7:0]  b_next,d_next;
reg      btn_reg;
wire db_tick,btn_tick,clr;
// 例化7 段LED 多路显示时间电路模块
disp_hex_mux disp_unit
(   .clk (clk),.reset(reset),
    .hex3(b_reg[7:4]),.hex2(b_reg[3:0]),
    .hex1(d_reg[7:4]),.hex0(d_reg[3:0]),
    .dp_in(4'b1011),  .an(an),.sseg(sseg));
// 例化循环跳转电路模块
debounce db_unit
(.clk(clk),.reset(reset),.sw(btn[1]),
zs.db_level(),.db_tick(db_tick));
// 对未翻转的输入进行边沿监测的电路
always @ (posedge clk)
btn_reg <= btn[1];
assign btn_tick = ~btn_reg & btn[1];
// 两个计数器
assign clr = btn[0];
always @ (posedge clk)
    begin
        d_reg <= d_next;
        b_reg <= b_next;
    end
// 计数器的下一逻辑状态
assign b_next = (clr)? 0:
                (btn_tick)? b - reg + 1:
```

```
              b_reg;
assign d_next = (clr) ? 0:
              (db_tick) ? d_reg + 1:
              d_reg;
endmodule
```

6.3 设计实例

6.3.1 斐波纳契数电路

斐波纳契数字构成了一个序列，定义如下：

$$fib(i)=\begin{cases}0 & \text{如果}\quad i=0\\ 1 & \text{如果}\quad i=1\\ fib(i-1)+fib(i-2) & \text{如果}\quad i>2\end{cases}$$

一种计算 $fib(i)$ 的方法是反复调用函数，从 0 到要求的值 i。这一方法需要两个临时寄存器，用于存储两个最近计数值[即 $fib(i-1)$ 和 $fib(i-2)$]，以及一个索引寄存器用于追踪迭代数字。如图 6-9 所示为 ASMD 图，t1 和 t0 为临时寄存器，n 为索引寄存器。除了常规数据输入/输出信号 i 和 f 外，我们还要包括一个用于指示开始的命令信号 start 以及两个状态信号：一个是 ready，用于指示电路是空闲状态并可以采样新的输入值；另一个是结束指示信号，用于判断操作完成是否经历了一个时钟周期。类似许多其他 FSMD 的设计，由于这种电路有可能是一个庞大系统设计的一部分，这些信号需要跟其他子系统进行接口。

ASMD 图有三种状态。idle 状态表示当前电路处于空闲。当 start 信号有效时，FSMD 跳转为 op 状态并载入初始值至 3 个寄存器。t0 和 t1 寄存器分别被载入 0 和 1，分别代表 $fib(0)$ 和 $fib(1)$。寄存器 n 载入预期的迭代次数 i。

主要的运算通过 3 次 RT 操作反复进入 op 状态来实现：

- t1←t1 + t0；
- t0←t1；
- n←n - 1。

前两次 RT 操作获取一个新值并将两个最近计算得到的 t1 和 t0 值存储。第三次 RT 操作将迭代索引减 1。当 n 值为 1 或者它的初始值为 0[即 $fib(0)$]时，停止迭代。与常规的流程图不同，在一个 ASMD 块中的所有操作可以在同一个时钟周期并行执行。我们将所有的比较和 RT 操作都放入 op 状态中，以减少计算时间。注意 t1 和 t0 的寄存器中的最新值在当 FSMD 退出 op 状态（即时钟的下一

个上升沿）时同时被更新。这样，t1 的原始值，而非 t1 + t0，被存储在 t0 中。done 状态的作用是生成一个单时钟周期信号 done-tick 用来指示计算完成。如果这个状态信号不需要，则这个状态可以被省略。

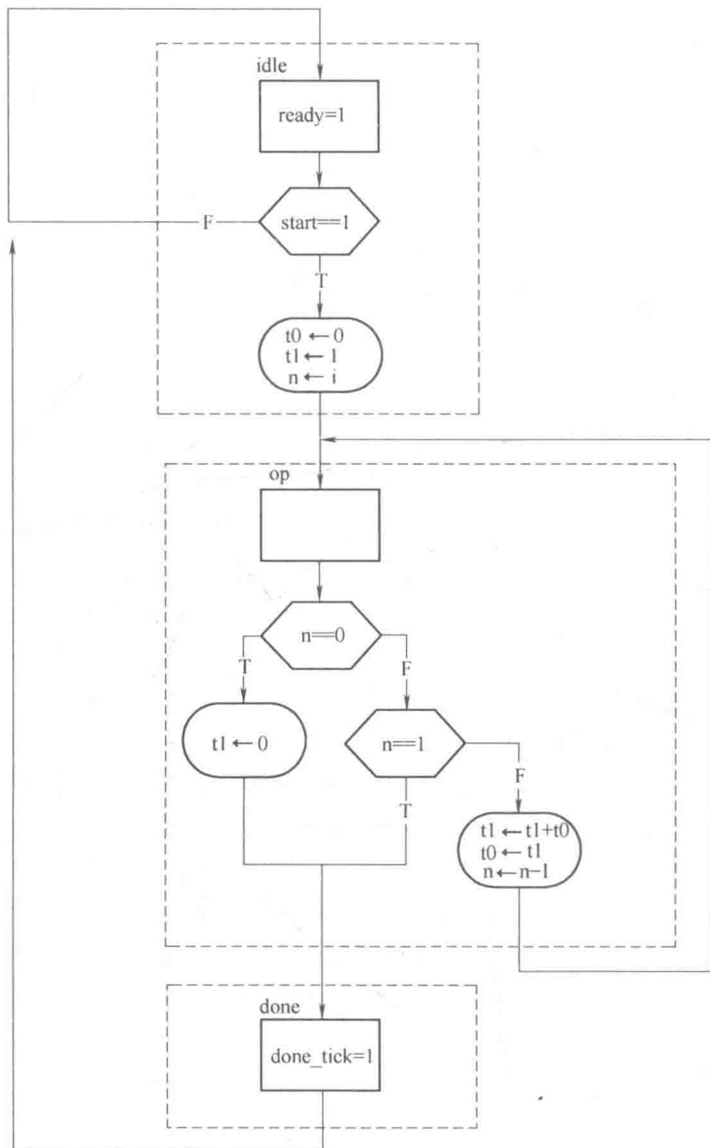


图 6-9　斐波纳契电路 ASMD 图

代码按 ASMD 图进行设计，如示例 6.4 所示。注意，斐波纳契函数增长迅速，输出信号的位宽要足够宽以适应期望结果。

示例 6.4　斐波纳契数电路

```
module fib
  (
    input wire clk, reset,
    input wire start,
    input wire [4:0] i,
    output reg ready, done_tick,
    output wire [19:0] f
  );
  // 符号化状态声明
  localparam [1:0]
      idle  =2'b00,
      op    =2'b01,
      done=2'b10;
  // 信号声明
  reg [1:0] state_reg, state_next;
  reg [19:0] t0_reg, t0_next, t1_reg, t1_next;
  reg [4:0] n_reg, n_next;
  // 代码部分
  //FSMD 状态和数据寄存器
  always @(posedge clk, posedge reset)
      if (reset)
        begin
            state_reg <= idle;
            t0_reg <=0;
            t1_reg <=0;
            n_reg <=0;
        end
      else
        begin
            state_reg <= state_next;
            t0_reg <= t0_next;
            t1_reg <= t1_next;
            n_reg <= n_next;
```

```
      end
//FSMD 下一状态逻辑
always @*
begin
    state_next = state_reg;
    ready = 1'b0;
    done_tick = 1'b0;
    t0_next = t0_reg;
    t1_next = t1_reg;
    n_next = n_reg;
    case (state_reg)
      idle:
          begin
            ready = 1'b1;
            if (start)
                begin
                  t0_next = 0;
                  t1_next = 20'd1;
                  n_next = i;
                  state_next = op;
                end
          end
      op:
          if(n_reg == 0)
            begin
                t1_next = 0;
                state_next = done;
            end
          else if(n_reg == 1)
            state_next = done;
          else
            begin
                t1_next = t1_reg + t0_reg;
                t0_next = t1_reg;
                n_next = n_reg - 1;
```

```
            end
        done:
            begin
                done_tick = 1'b1;
                state_next = idle;
            end
        default: state_next = idle;
      endcase
   end
   // 输出
   assign f = t1_reg;

endmodule
```

6.3.2　除法电路

由于除法操作的复杂性，除法运算不能被直接自动综合。在这一章节中我们通过使用 FSMD 来实现长除法运算。算法通过两个 4bit 无符号整型数的除法来说明，如图 6-10 所示。算法总结如下：

1）通过在高位添加“0”的方式来双倍扩展被除数位宽，并将除数与扩展后被除数最左位对齐；

2）如果对应的被除数位宽大于或者等于除数，则从被除数中减去除数并将对应商值位置 1。否则，保持原始除数位，并将对应商值位置 0。

3）在上一次结果中增加一个额外的被除数位并且将除数右移动一位；

4）重复步骤 2 和 3 直到所有被除数位数都被除尽。

```
除数
 /          00110 — 商
0010 /00001101 — 被除数
      0000
       0001
       0000
        0011
        0010
         0010
         0010
          0001 — 余数
```

图 6-10　两个 4bit 无符号整型数长除法

数据路径的结构如图 6-11 所示。首先，除数被放入寄存器 d 并且经过扩展的被除数被放入寄存器 rh 和 rl。在每次循环中，rh 和 rl 寄存器被左移一位。除数在原先的运算结果上向右进行相应的移动。如果寄存器 rh 中的值（余数）远远大于或者等于寄存器 d 中的值（除数），我们就将 rh 和 d 中的值进行比较相减。当 rh 和 rl 中的值左移到相应的位置，rl 中最右端的比特位数变为可用。可以用来存储当前的商数比特。当我们将所有的被除数位宽除尽时，最后不能整除的即为余数存储在寄存器 rh 中，所有的商数位宽都移入寄存器 rl。

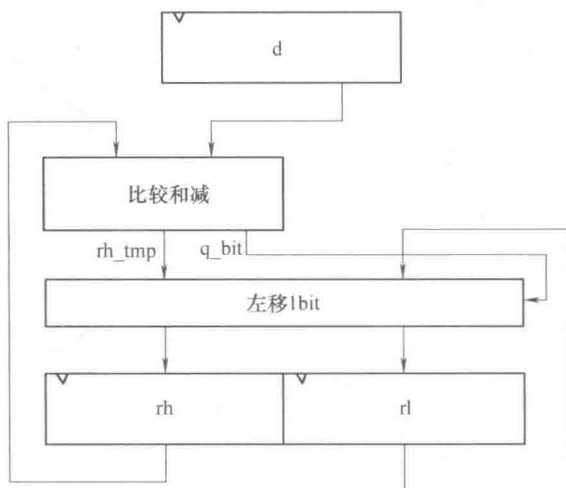


图 6-11　除法电路的数据流向缩略图

除法电路的 ASMD 图与之前的斐波纳契电路有些相似。FSMD 由 4 个状态组成：idle、op、last 和 done。为了代码清晰起见，我们将比较和减法电路从代码片段中分离出来。主要的运算功能在 op 状态中实现，被除数和除数在这个状态中进行比较并相减然后左移一位。注意，在最后一次迭代运算中余数不应被移位。我们创建了一个单独的状态 last，以适应这个特殊要求。与之前的例子一样，done 状态的作用是用来生成一个单时钟周期的 done-tick 信号用来指示计算的完成。代码如示例 6.5 所示。

示例 6.5　除法电路

```
module div
  # (
    parameter W = 8,
              CBIT = 4   // CBIT = log2 (W) + 1
  )
   (
    input wire clk, reset,
    input wire start,
    input wire [W - 1: 0] dvsr, dvnd,
    output reg ready, done_tick,
    output wire [W - 1: 0] quo, rmd
  );
  // 符号化状态定义
```

```
localparam [1: 0]
    idle =2' b00,
    op   =2' b01,
    last =2' b10,
    done=2' b11;
//信号声明
reg [1: 0] state_reg, state_next;
reg [W -1: 0] rh_reg, rh_next, rl_reg, rl_next, rh_tmp;
reg [W -1: 0] d_reg, d_next;
reg [CBIT -1: 0] n_reg, n_next;
reg q_bit;
//代码部分
//FSMD 状态和数据寄存器
always @ (posedge clk, posedge reset)
    if (reset)
      begin
          state_reg <= idle;
          rh_reg <=0;
          rl_reg <=0;
          d_reg <=0;
          n_reg <=0;
      end
    else
      begin
          state_reg <= state_next;
          rh_reg <= rh_next;
          rl_reg <= rl_next;
          d_reg <= d_next;
          n_reg <= n_next;
      end
//FSMD 下一逻辑状态
always @ *
begin
    state_next = state_reg;
    ready =1' b0;
```

```
done_tick = 1'b0;
rh_next = rh_reg;
rl_next = rl_reg;
d_next = d_reg;
n_next = n_reg;
case (state_reg)
  idle:
      begin
        ready = 1'b1;
        if (start)
            begin
              rh_next = 0;
              rl_next = dvnd;     // 被除数
              d_next = dvsr;      // 除数
              n_next = CBIT;      // 指示位
              state_next = op;
            end
      end
  op:
      begin
        // 左移rh 和rl
        rl_next = {rl_reg [W-2: 0], q_bit};
        rh_next = {rh_tmp [W-2: 0], rl_reg [W-1]};
        // 指数下降
        n_next = n_reg - 1;
        if (n_next ==1)
        state_next = last;
      end
  last: // 最后一次迭代
      begin
        rl_next = {rl_reg [W-2: 0], q_bit};
        rh_next = rh_tmp;
        state_next = done;
      end
  done:
```

```
                begin
                  done_tick = 1'b1;
                  state_next = idle;
                end
            default: state_next = idle;
        endcase
    end
    // 比较和减法电路
    always @ *
        if (rh_reg >= d_reg)
          begin
              rh_tmp = rh_reg - d_reg;
              q_bit = 1'b1;
          end
        else
          begin
              rh_tmp = rh_reg;
              q_bit = 1'b0;
          end
    // 输出
    assign quo = rl_reg;
    assign rmd = rh_reg;
endmodule
```

6.3.3　二进制向 BCD 码转换电路

在 4.5.2 节我们讨论了 BCD 码的格式。在这种格式下，一个十进制数可以用一个 4 位的 BCD 数序列表示。一个二进制向 BCD 码转换电路将一个二进制数转换为 BCD 码形式。例如，二进制数"0010 0000 0000"经过转换变为"0101 0001 0010"（即 512_{10}）。

二进制码向 BCD 码的转换可以通过一个专门的 BCD 移位寄存器处理，它将二进制数分成 4 位一组，每组代表一个 BCD 数。如果向 BCD 数比移动后的小，向左移动的 BCD 序列就需要调整。举例来说，如果一个 BCD 码序列是"0001 0111"（即 17_{10}），它应该变为"0011 0100"（即十进制的 34），而不是"0010 11 10"。这种调整需要从右边的 BCD 码中减去一个十进制的 10（即

“1010”），然后对下一个 BCD 码加 1（可以看作是进位）。需要注意的是，对于一个 4 位宽的二进制数来说，减去一个十进制的数值 10 相当于加上一个十进制的数值 6。这样，前述调整也可通过在正确的 BCD 码上增加十进制的 6 完成。该进位为进程中自动生成。

在实际的实现中，首先对 BCD 码进行必要的调整然后再移位更为高效。可以检查一个 BCD 码是否大于十进制的数码 4，如果是，便在这个数值上再加一个十进制的 3。当所有的 BCD 码经过校正以后，便可以将整个寄存器向左移动一位。一个二进制向 BCD 码转换电路可以通过将二进制输入按位从 MSB 到 LSB 移位构建。操作总结如下：

1）对于 BCD 移位寄存器中 4 位宽的 BCD 码的每位数字，检查是否大于 4。如果是，则加十进制 3；

2）将整个 BCD 码寄存器左移 1 位，并且将输入二进制数的 MSB 移入 BCD 寄存器的 LSB 中；

3）重复步骤 1、2 直到所有的输入位宽全部使用。

7bit 位宽的二进制输入“1111111”（即十进制的 127）的转换过程见表 6-1。

表 6-1　二进制码向 BCD 码转换实例

操　作		专用 BCD 码移位寄存器			二进制码输入
		BCD 码 2	BCD 码 1	BCD 码 0	
初始化					111 1111
第 6 比特	无调整 左移 1 位			1 (1_{10})	11 1111
第 5 比特	无调整 左移 1 位			11 (1_{10})	1 1111
第 4 比特	无调整 左移 1 位			11 (3_{10})	1111
第 3 比特	BCD 数字 0 调整 左移 1 位		1 (1_{10})	1010 0101 (5_{10})	111
第 2 比特	BCD 数字 0 调整 左移 1 位		1 11 (3_{10})	1000 0001 (1_{10})	11
第 1 比特	无调整 左移 1 位		110 (6_{10})	0011 (3_{10})	1
第 0 比特	BCD 数字 1 调整 左移 1 位	1 (1_{10})	1001 0010 (2_{10})	0011 0111 (7_{10})	

13 位宽转换电路代码如示例 6.6 所示。它使用了一个简单的 FSMD 来控制所有的操作。当 start 信号被置为有效时，二进制输入被存储在 p2s 寄存器中。与前述例子中的进程描述相似，状态机反复迭代 13bit 的输入数据。4 个调整电路被用来修正 4 个 BCD 码。为了清晰起见，它们从次态逻辑中被分离了出来，并用单独的代码段进行了描述。

示例 6.6　二进制码向 BCD 码转换电路

```
module  bin2bcd
(
input wire clk,reset,
input wire start,
input wire [12:0] bin,
output reg ready,done_tick,
output wire [3:0] bcd3,bcd2,bcd1,bcd0
);
// 状态符号化定义
localparam [1: 0]
    idle  =2 'b00,
    op   =2'b01,
    done  =2 'b10;
// 信号定义
reg  [1: 0]  state_reg ,state_next;
reg  [12:0]  p2_sdreg,p2s_next;
reg  [3: 0]  n_reg ,n_next;
reg  [3: 0]  bcd3_reg,bcd2_reg,bcd1_reg ,bcd0_reg;
reg  [3:0]  bcd3_next ,bcd2_next ,bcd1_next ,bcd0_next;
wire [3:0]  bcd3_tmp,bcd2_tmp ,bcd1_tmp ,bcd0_tmp;
// 代码主体
//FSMD 状态和数据寄存器
always Q (posedge clk ,posedge reset)
    if (reset)
      begin
        state_reg <= idle;
        p2s_reg  <=0;
        n_reg  . <=0;
```

```
        bcd3_reg  <=0;
        bcd2_reg  <=0;
        bcd1_reg  <=0;
        bcd0_reg  <=0;
      end
    else
    begin
        state_reg  <=state_next;
        p2s_reg    <=p2s_next;
        n_reg      <=n_next;
        bcd3_reg  <=bcd3_next;
        bcd2_reg  <=bcd2_next;
        bcd1_reg  <=bcd1_next;
        bcd0_reg  <=bcd0_next;
      end
//FSMD 下一状态逻辑
always @ *
    begin
      state_next =state_reg;
      ready =1'b0;
      done_tick =1'b0;
      n_next =n_reg;
      case (state_reg)
      idle :
      begin
        ready =1'b1;
        if (start)
          begin
            state_next =op;
            bcd3_next =0;
            bcd1_next =0;
            bcd0_next =0;
            n_next =4'b1101; //索引
            p2s_next =bin; //移位寄存器
            state_next =op;
```

```
            end
          end
          op:
            begin
              // 二进制码移位
              p2s_next = p2s_reg  < < 1;
//4 位二进制码移位
// { bcd3_next , bcd2_next , bcd1_next , bcd0_next  ) }
// { bcd3_tmp  [2 : 0 ], bcd2_tmp , bcd1_tmp , bcd0_tmp ,
//p2s_reg  [12 ] }
              bcd0_next = {bcd0_tmp [2: 0] ,p2s_reg [12]}
              bcd1_next = {bcd1_tmp [2:0] ,bcd0_tmp C311}
              bcd2_next = {bcd2_tmp [2:0] ,bcdl_tmp [31]}
              bcd3_next = {bcd3_tmp [2:0] ,bcd2_tmp [31]}
              n_next    = n_reg_1;
              if (n_next = =0)
                state_next = done;
              end
            done:
              begin
                done_tick = 1'b1;
                state_next = idle;
              end
          default:state_next = idle;
        endcase
      end
// 数据流功能单元
assign bcd0_tmp = (bcd0_reg  > 4) ? bcd0_reg +3  : bcd0_reg;
assign bcd1_tmp = (bcd1_reg  > 4) ? bcd1_reg +3  : bcd1_reg;
assign bcd2_tmp = (bcd2_reg  > 4) ? bcd2_reg +3  : bcd2_reg;
assign bcd3_tmp = (bcd3_reg  > 4) ? bcd3_reg +3  : bcd3_reg;
// 输出
assign bcd0 = bcd0_reg;
assign bcd1 = bcd1_reg;
assign bcd2 = bcd2_reg;
```

```
assign bcd3 = bcd3_reg;
endmodule
```

6.3.4 周期计数器

周期计数器用于测量周期性的输入波形的周期。一种构造方式是计算输入信号的两个上升沿之间的时钟周期个数。由于系统时钟的频率是已知的，因此输入信号的周期可以被算出。例如，如果系统时钟周期是 f 而在两个上升沿之间的时钟周期个数是 N，可以推断出输入信号的周期是 $N*\frac{1}{f}$。

在这一小节中，设计的测量精度是毫秒级。ASMD 图如图 6-12 所示。当 start 信号有效时，周期计数器开始执行测量。我们使用一个上升沿检测电路来生成一个单时钟周期的标识信号，表示输入波形信号的上升沿。start 信号有效后，FSMD 跳转到 waite 状态，等待输入信号的第一个上升沿。当检测到输入的第二个上升沿时，FSMD 进入 count 状态。在 count 状态中，我们使用两个寄存器来追踪时间。t 寄存器可以计算 50，000 个时钟周期，从 0 ~ 49，999 再重新开始。由于系统的时钟周期是 20ns，t 寄存器用 50，000 个时钟周期来计算 1ms。p 寄存器以 ms 为单位进行计数，每当 t 寄存器到达 49，999 时加 1。当 FSMD 退出 count 状态，输入波形的周期被存入 p 寄存器且它的单位是 ms。与前面例子一样，FSMD 在 done 状态时置 done_ tick 信号有效。

程序代码在示例 6.7 中给出。使用一个常量 CLK_MS_COUNT 作为毫秒计数器的边界值，如果需要使用另一个测量单位，则它可以被替换。

示例 6.7　周期计数器

```
module period_counter
  (
    input wire clk,reset,
    input wire start,si,
    output reg ready,done_tick,
    output wire [9:0]prd
  );
  //状态符号化定义
localparam [1:0]
       idle  =2'b00,
       waite =2'b01,
```

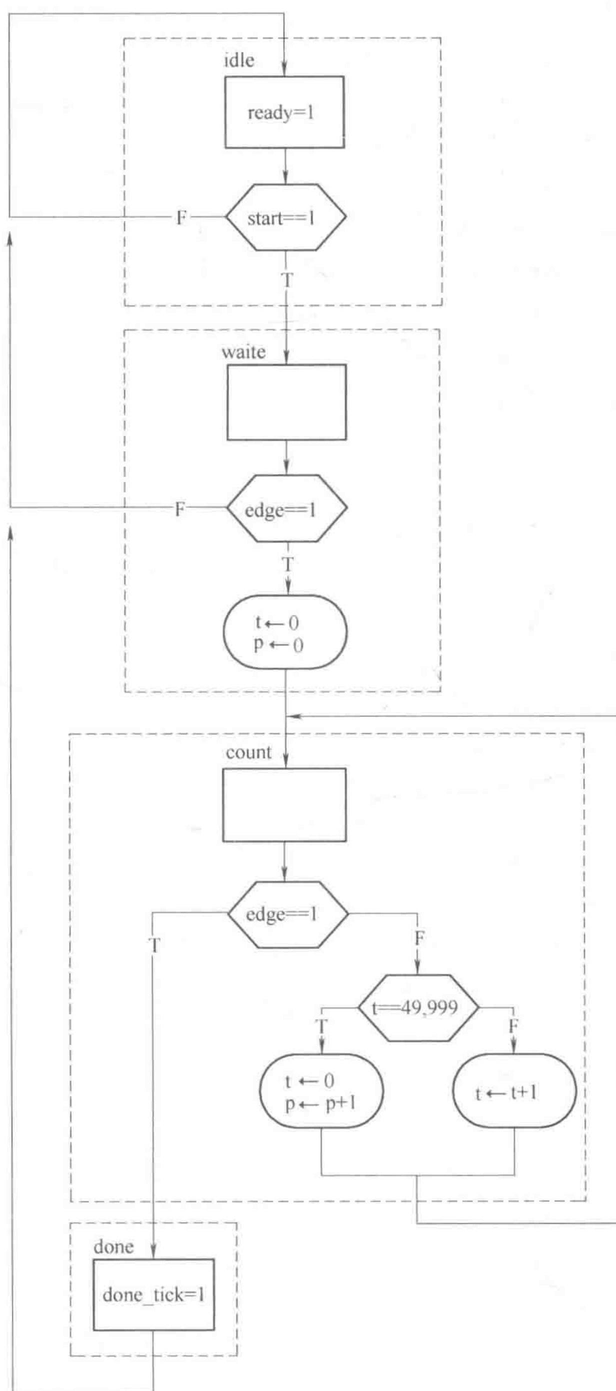


图 6-12　周期计数器的 ASMD 图

```
        count =2'b10,
        done  =2'b11;
  // 常量声明
localparam CLK_MS_COUNT =50000; // 计时
  // 信号声明
  reg [1:0] state_reg,state_next;
reg [15:0] t_reg,t_next;   // 最大值50000
reg [9:0] p_reg,p_next;    // 最大值1 秒
  reg delay_reg;
  wire edg;
  // 实体
  //FSMD 状态和数据寄存器
  always @ (posedge clk,posedge reset)
        if (reset)
          begin
              state_reg  <= idle;
              t_reg  <=0;
              p_reg  <=0;
              delay_reg  <=0;
          end
        else
          begin
              state_reg  <= state_next;
              t_reg  <=t_next;
              p_reg  <=p_next;
              delay_reg  <=si;
          end
  // 上升沿 tick
  assign edg = ~delay_reg & si;
  //FSMD 下一状态逻辑
  always @ *
  begin
        state_next = state_reg;
        ready =1'b0;
        done_tick =1'b0;
```

```
    p_next = p_reg;
    t_next = t_reg;
    case (state_reg)
      idle:
          begin
              ready = 1'b1;
              if (start)
                state_next = waite;
          end
      waite: // 等待第一个沿触发
          if (edg)
            begin
                state_next = count;
                t_next = 0;
                p_next = 0;
            end
      count:
          if (edg)// 第二个沿到达
            state_next = done;
          else// 否则计数
            if (t_reg = = CLK_MS_COUNT - 1) //1 ms tick
                begin
                  t_next = 0;
                  p_next = p_reg + 1;
                end
            else
                t_next = t_reg + 1;
      done:
          begin
            done_tick = 1'b1;
            state_next = idle;
          end
      default: state_next = idle;
    endcase
end
```

```
  // 输出
  assign prd = p_reg;
endmodule
```

6.3.5 精确的低频计数器

频率计数器可以测量一个周期性输入波形的频率。通常的方法是构建一个频率计数器在一段固定的时间内计算输入脉冲的个数，比如说 1s。尽管这种方法对于高频输入来说还不错，但这种方法无法精确地测量一个低频信号。例如，如果输入是 2Hz 左右，测量便无法准确区分是 2.213Hz 还是 2.567Hz。回忆一下，频率是周期的倒数 $\left(\text{即频率} = \frac{1}{\text{周期}}\right)$。另一个方案是测量信号的周期然后取倒数求得频率。在这一小节中我们使用这种方法来实现一个低频率计数器。

这个设计展示了如何使用一个预先设计好的部分来构建一个更大的系统。为简单起见，我们假设输入的频率在 1 ~ 10Hz（周期在 100 ~ 1000ms 之间）。电路的运行包含 3 个任务：

1）测量周期；

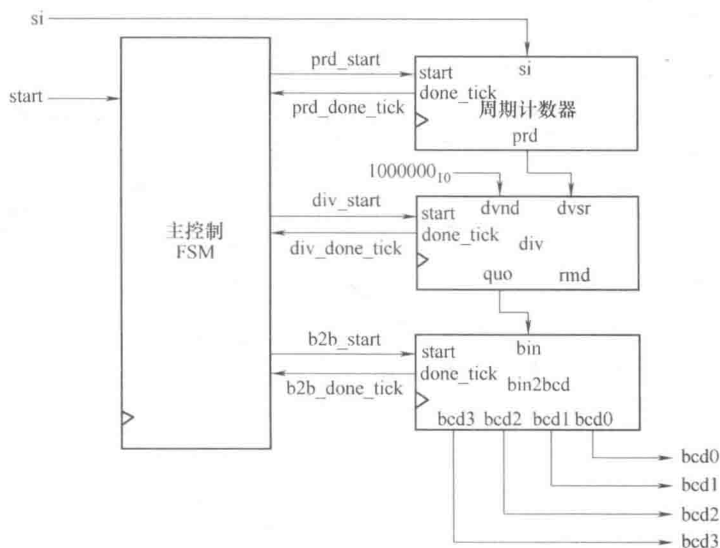
2）通过实现除法运算来求得频率；

3）转换二进制码为 BCD 码形式。

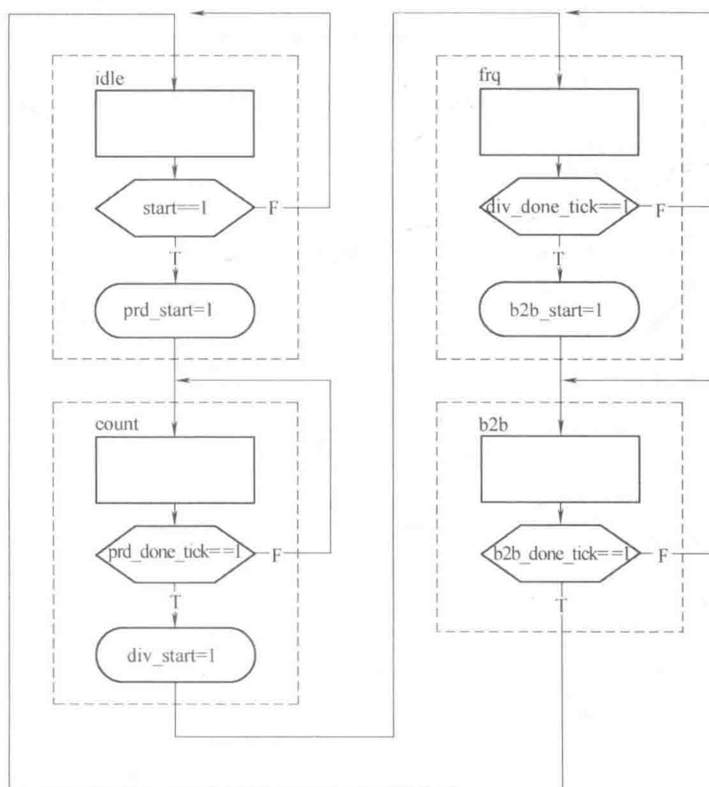
我们可以使用周期计数器、除法电路以及二进制向 BCD 码转换电路来实现这三种任务并且通过创建另一个 FSM 作为主控，控制协调 3 个电路的运行。电路模块框图如图 6-13a 所示，主控的 ASM 图如图 6-13b 所示。FSM 电路的 start 信号和 done_tick 信号用来初始化每个任务和监测任务是否执行完毕。代码如示例 6.8 所示。

示例 6.8　低频率计数器

```
module low_freq_counter
  (
    input wire clk, reset,
    input wire start, si,
    output wire [3:0] bcd3, bcd2, bcd1, bcd0
  );
  // 状态符号化定义
  localparam  [1:0]
              idle    = 2'b00,
```



a) 顶层模块图



b) ASM表的主要控制

图 6-13　低频精确率计数器

```
            count =2'b01,
            frq   =2'b10,
            b2b   =2'b11;
// 信号声明
reg [1:0] state_reg,state_next;
wire [9:0]prd;
wire [19:0]dvsr,dvnd,quo;
reg prd_start,div_start,b2b_start;
wire prd_done_tick,div_done_tick,b2b_done_tick;
// ===========================================
// 器件例化
// ===========================================
// 例化周期计数器
period_counter prd_count_unit
    (.clk(clk),.reset(reset),.start(prd_start),.si(si),
    .ready(),.done_tick(prd_done_tick),.prd(prd));
// 例化除法电路
div #(.W(20),.CBIT(5)) div_unit
    (.clk(clk),.reset(reset),.start(div_start),
    .dvsr(dvsr),.dvnd(dvnd),.quo(quo),.rmd(),
    .ready(),.done_tick(div_done_tick));
// 例化二进制码向BCD 码转换器
bin2bcd b2b_unit
    (.clk(clk),.reset(reset),.start(b2b_start),
    .bin(quo[12:0]),.ready(),.done_tick(b2b_done_tick),
    .bcd3(bcd3),.bcd2(bcd2),.bcd1(bcd1),.bcd0(bcd0));
// 信号位宽拓展
assign dvnd =20'd1000000;
assign dvsr ={10'b0,prd};
// ===========================================
// 主状态机
// ===========================================
always @ (posedge clk,posedge reset)
   if (reset)
        state_reg <=idle;
```

```
    else
        state_reg <= state_next;
  always @ *
  begin
      state_next = state_reg;
      prd_start = 1'b0;
      div_start = 1'b0;
      b2b_start = 1'b0;
      case (state_reg)
        idle:
            if (start)
              begin
                  prd_start = 1'b1;
                  state_next = count;
              end
        count:
            if (prd_done_tick)
              begin
                  div_start = 1'b1;
                  state_next = frq;
              end
        frq:
            if (div_done_tick)
              begin
                  b2b_start = 1'b1;
                  state_next = b2b;
              end
        b2b:
            if (b2b_done_tick)
              state_next = idle;
        endcase
  end
endmodule
```

6.4　文献备注

FSMD 通常会在高层次综合的背景下讨论。D. D. Gajsk 的《high - level synthesis. Principles of Digital Design》一书中包含了一个全的章节，讨论了相关的内容和 FSMD 算法的设计和实现。

6.5　实验

6.5.1　另一种去抖电路

思考 5.5.2 节实验中的跳转电路。使用 RT 方法学重新设计电路：

1）设计电路的 ASMD 图；

2）设计基于 ASMD 图的 HDL 代码；

3）用新的设计替换 6.2.5 节中的去抖电路，并验证其操作。

6.5.2　BCD 码向二进制码转换电路

一个 BCD 码向二进制码转换电路将一个 BCD 数转换为一个与其相等的二进制数表示。假设输入信号是一个 8bit 的 BCD 码格式信号（也就是两个 BCD 码），输出是一个 7bit 的二进制表示信号。按照 6.3.3 节中的步骤设计一个 BCD 码向二进制码转换电路：

1）设计转换算法和 ASMD 图；

2）设计基于 ASMD 图的 HDL 代码；

3）设计测试平台并且通过仿真验证代码运行；

4）综合电路，对 FPGA 进行编程并验证其运行。

6.5.3　带有 BCD I/O 的斐波纳契电路：设计方法 I

为了使斐波纳契电路的应用更为友好，我们可以修改电路，使用 BCD 码输入和输出。假设输入是一个 8bit 的 BCD 码信号（也就是两个 BCD 码）而输出用 BCD 码在七段 LED 数码管显示。然而，当斐波纳契数列的结果大于 9999（也就是数值越界）时，LED 数码管将会显示“9999”。这种运算可以分为三步来执行：①将输入转换为二进制编码形式；②计算斐波纳契数字；③将结果转换回 BCD 格式。

第一种设计方法遵循 6.3.5 节提供的程序步骤。首先创建 3 个小的子系统，分别是 BCD 码向二进制码转换电路，斐波纳契电路，二进制向 BCD 码转换电

路，然后使用一个主状态机来控制所有的操作。电路的设计流程如下所示：

1）实现实验6.2.5中BCD码向二进制码转换的电路；

2）修改6.3.1节中的斐波纳契电路，增加一个用于显示越界的输出信号；

3）设计顶层框图和主控FSM状态图；

4）设计HDL代码；

5）设计测试平台并用于仿真验证代码的运行；

6）综合并形成电路，在FPGA上编程下载，并验证FPGA的运行。

6.5.4　带有BCD I/O的斐波纳契电路：设计方法Ⅱ

另外一种代替实验6.5.3中的“子系统方案”的设计方法是将3个子系统集成为一个系统并为这种特定的应用设计一个定制的FSMD。这一方法消除了控制状态机的管理，并为这3种任务的寄存器共享提供了机会。设计电路过程如下：

1）使用一个FSMD重新设计实验6.5.3的电路，设计方应去除所有多余的电路和状态，例如各种各样的done-tick信号和done状态，同时拓展机会用于共享和重用不同步骤中的寄存器；

2）设计ASMD图；

3）设计基于ASMD图的代码；

4）设计测试平台并用于仿真验证代码的运行；

5）综合电路，并下载至目标FPGA芯片中，验证运行；

6）检查综合报告并比较这两种方式的LEs的利用率；

7）计算两种方式完成操作所需要的时钟周期数。

6.5.5　尺度自适应的低频计数器

6.3.5节中的低频率计数器的操作非常受限。输入信号的频率范围被限制在了1~10Hz之间。当超出这个范围时它将失去准确性。回忆6.3.5节，频率计数器的精度依赖于周期计数器的精度，周期计数器计数到ms级tick。我们可以修改t计数器以生成一个μs级tick（即从0~49间计数），将精度提高1000倍。这一方法可以使频率计数器计数范围增加到9999Hz，仍然保持至少4位精确性。

对于低频输入来说使用一个μs级的tick可以带来大于4位的精度，并且数值必须被移位和截断以便于显示在七段LED数码管上。一个尺度自适应的低频计数器会自动调整运行，显示最左边的4个数，同时在合适的位置放置十进制小数点。例如，根据不同的范围，频率测量值将会显示为“1.234”，“12.34”，“123.4”或者“1234”。

这种自适应范围的低频率计数器需要一个BCD码调整电路。它首先检查

BCD 序列最重要的 BCD 码（也就是 4 个最高有效位）的值是否为零。如果为零，电路将按顺序向左移动 BCD 位置并且对十进制分数进行计数。这种操作是反复执行的，直到最关键的 BCD 码不再是“0000”。

完整的尺度自适应低频计数器可以按照如下步骤来实现：

1）使用 ms 级 tick 修改周期计数器；

2）扩展二进制码到 BCD 码转换电路的数值范围；

3）设计 BCD 码调整电路和代码的 ASMD 图；

4）在最后一步修改控制状态机，将 BCD 码调整包含进去；

5）设计一个简单的译码电路，使用小数点计数器的输出去激活七段 LED 数码管显示期望的小数点；

6）设计测试平台并且通过仿真来验证代码功能；

7）综合电路，对 FPGA 芯片进行编程并验证其功能。

6.5.6　反应定时电路

手眼协调性是指双手和眼睛协同工作完成一个任务时的能力。一个反应定时电路用于测量一个人在看到一个可视的外界激励的时候做出反应的速度。这种电路运行如下：

1）电路有 3 个输入按钮，对应 clear、start 和 stop 信号，电路使用一个单一的分立 LED 作为视觉刺激，并在七段 LED 数码管上显示相关信息；

2）使用者通过按下“clear”按钮强制使电路返回初始状态，此时七段数码管显示一条欢迎信息“HI”，视觉刺激 LED 关闭；

3）当准备就绪后，使用者将按下“start”按钮以初始化为测试状态，七段数码管熄灭；

4）经过一个 2 ~ 15s 的随机时间间隔之后，视觉刺激 LED 点亮，同时计时器开始向上计数。计时器每隔 1ms 加 1 并且它的值将以“0.0000” s 的形式显示在七段 LED 数码管上；

5）当视觉激励 LED 灯亮后，使用者应尽可能快地按下“stop”按钮，一旦停止按钮有效时计时器将停止计数，七段数码管将显示响应时间，对于绝大多数人来说它的值应该在 0.15 ~ 0.30s 之间；

6）如果没有按下“stop”按钮，计时器在 1s 后停止并且显示“1.000”；

7）如果“stop”按钮在视觉刺激 LED 点亮之前按下，电路将停止并在七段数码管显示“9.999”。

设计电路步骤如下：

1）设计 ASMD 图；

2）根据 ASMD 图设计 HDL 代码；

3）综合电路，对 FPGA 芯片进行编程并验证其功能。

6.5.7　巴贝奇差分引擎模拟电路

巴贝奇差分引擎是一个对多项式函数进行列表的机械式数字计算设备。由巴比奇·查尔斯提出，一位 19 世纪的英国数学家。该引擎基于牛顿的差分计算方法从而避免了乘法器的出现。例如，考查一个二阶多项式 $f(n)=2n^2+3n+5$。我们可以得到 $f(n)$ 与 $f(n-1)$ 的差：

$$f(n)-f(n-1)=4n+1$$

假设 n 为整数且 $n\geqslant 0$。$f(n)$ 可以被递归定义为

$$f(n)=\begin{cases}5 & \text{如果}\quad n=1\\ g(n-1)+4n+1 & \text{如果}\quad n>1\end{cases}$$

这个过程可以用 $4n+1$ 表达式进行重复。设 $g(n)=4n+1$。可以找到 $g(n)$ 与 $g(n-1)$ 的差：

$$g(n)-g(n-1)=4$$

$g(n)$ 可以被递归定义为

$$g(n)=\begin{cases}5 & \text{如果}\quad n=0\\ g(n-1)+4 & \text{如果}\quad n>1\end{cases}$$

且 $f(n)$ 可以被重新写为

$$f(n)=\begin{cases}5 & \text{如果}\quad n=0\\ f(n-1)+g(n) & \text{如果}\quad n>0\end{cases}$$

需要注意的是，在递归定义的 $f(n)$ 和 $g(n)$ 中只涉及了加法运算。

从最后的两个递归等式的定义可以得出计算 $f(n)$ 的算法。需要两个临时寄存器来追踪并记录最近计算所得的 $f(n)$ 和 $g(n)$ 的值，以及另外两个寄存器来更新 $f(n)$ 和 $g(n)$。假设 n 是一个 6 位的输入并且当作一个无符号整数。使用 RT 方法学设计这个电路：

1）设计 ASMD 图；

2）设计基于 ASMD 图的 HDL 代码；

3）设计测试平台并且通过仿真验证代码功能；

4）综合源代码为实际电路，对 FPGA 进行编程，验证 FPGA 的功能；

5）假设 $h(n)=n^3+2n^2+2n+1$，使用上述方法得出 $h(n)$ 的递归表达式（注意，对于一个三次多项式来说需要 3 个层次的递归等式）。重复步骤 1～4。

第7章　Verilog相关的话题

由于本书主要关注数字设计，因此，我们只介绍Verilog语言的最小子集，并依赖于比较简单的设计原则和模版。在这一章，我们就Verilog语言选择了一些比较重要的话题进行深入的讨论。除了上一部分对仿真相关的构建方式进行了简述以外，这些话题都与综合相关，且能帮助我们开发更加复杂的代码。也可以跳过本章内容，不会对其他章节的理解产生影响。

7.1　阻塞和非阻塞

在always结构块中，有两种赋值方法可以使用：阻塞赋值和非阻塞赋值。3个简单的使用原则在前面的章节里已经给出：

- 将电路划分为寄存器电路和组合逻辑电路；
- 为寄存器电路选择一个合适的模版，对寄存器要使用非阻塞赋值；
- 使用阻塞赋值描述组合逻辑电路。

在这一部分，本书将分析这两种赋值方法，并解释使用原则背后两者间的关系，本书将在下一部分介绍另外一种可选的代码风格。

7.1.1　概述

1. 阻塞赋值　阻塞赋值的基本语法为

[变量]=[表达式]；

当赋值语句执行时，等式右边的表达式值被计算出来，并赋值给等式左边的变量，这一过程不会被其他任何语句所打断。这样，它“阻塞”了其他语句的执行，直到当前的赋值过程执行完毕。阻塞赋值的行为与C语言变量赋值相似。

2. 非阻塞赋值　非阻塞赋值的基本语法为

[变量] <=[表达式]；

非阻塞赋值的行为更加微妙，从硬件的角度去解释最合适不过了。回想一下，一个always块可被看作为一个抽象的硬件部件。时序控制结构可以添加到always块中用于模拟传播延迟。对于我们的可综合代码，当没有显示的时序控制时，会存在一个隐式的假定时间步长用于模拟延迟。当always块被激活时，等式右边的非阻塞表达式的值在时间步长的一开始便被计算出来。当执行到always块的最后时（即时间步长结束时），右边表达式的值赋值给等式左边的非阻塞赋

值变量。由于其他语句可以在求值和赋值过程中执行，因此这种赋值方式被称为“非阻塞”。

假设变量 x 通过非阻塞赋值方式被赋值。然而，Verilog的实际调度却相当复杂，非阻塞赋值的行为可以被解释成如下过程：

- 在always块的起始，x 被赋值给 x_{entry}；
- x_{exit} 代替等式左边变量 x；
- x_{entry} 代替等式右边变量 x；
- 在always块的结束，x_{exit} 赋值给 x。

对应的解释可参见下面代码中的注释部分：

```
always @ *
begin// x_entry = x
…
y <= x & …// y = x_entry & …
x <= …// x_exit = …
…
end// x = x_exit
```

示例：为了能够理解阻塞与非阻塞赋值间的区别，让我们思考一下在3.3.4节讨论过的3个输入和电路。相应代码重复放在示例7.1中。代码中使用的是阻塞赋值，推断出的电路如图3-3a所示。

示例7.1 使用阻塞赋值的与门

```
module and_block
(
  input wire a, b, c,
  output reg y
  );
  always @ *
  begin
      y = a;
      y = y & b;
      y = y & c;
  end
endmodule
```

阻塞赋值的行为与C语言的顺序语句相似，y 变量在最后得到 a、b、c 相与

的值。注意本段代码只是出于展示的目的。使用顺序语义描述硬件是非常不好的设计实践。

如果我们使用非阻塞赋值代替阻塞赋值，修改后的代码如示例 7.2 所示。注释代码是对使用 y 的解释。

示例 7.2　与门 - 使用非阻塞赋值

```
module and-nonblock
(
    input waire a,b,c,
    output reg y
);
  always @ *
  begin//   y_entry = y
    y  <= a; //   y_exit = a
    y  <= y & b;//   y_exit = y_entry & b
    y  <= y & c;//   y_exit = y_entry & c
  end//   y = y_exit
endmodule
```

注意，前两句赋值语句没有任何作用，与下面代码相同

```
always @ *
  y  <= y & c;
```

相应的电路图如图 3-3b 所示，并非为期望电路。

7.1.2　组合逻辑电路

前面小节中的例子是个极端的例子。除了默认值，绝大多数组合逻辑电路代码不会向同一变量多次赋值。对于同样的电路，阻塞赋值和非阻塞赋值均能实现正确描述。然而，它们之间还是存在着一些细微的差别。思考一下在 1.2 节讨论的 1bit 相等判断电路。使用阻塞赋值修改后的代码如示例 7.3 所示。我们在敏感列表中显示地列出了变量。

示例 7.3　使用阻塞赋值的相等判断电路

```
module eq1_block
  (
  input wire i0, i1,
```

```
  output reg eq
  );
  reg p0, p1;
  always @ (i0, i1) // 敏感列表中只有i0 和i1
  // 语句的顺序非常重要
  begin
    p0 = ~i0 & ~i1;
    p1 = i0 & i1;
    eq = p0 | p1;
  end
endmodule
```

注意，敏感列表中只包含 i0 和 i1。当其中任何一个发生变化时，always 块便会被激活。p0、p1 和 eq 被顺序计算求值。eq 在第一个时间步长的结束被更新。

语句的顺序非常重要。假设我们将最后一个语句挪到开始位置：

```
always @ (i0, i1)
begin
  eq = p0 | p1;
  p0 = ~i0 & ~i1;
  pl = i0 & i1;
end
```

由于 p0 和 p1 还没有被赋予新值，将会使用上次激活产生的值。使用上次的值便推断出了锁存器，因此这段代码是不正确的。

可以使用非阻塞赋值代替阻塞赋值。如示例 7.4 所示。赋值语句的解释参见代码的注释。

示例 7.4 使用非阻塞赋值的相等判断电路

```
module eq1_non_block
  (
  input wire i0, i1,
  output reg eq
  );
  reg p0, p1;
  always @ (i0, i1, p0, p1) //p0, p1 也在敏感列表中
```

```
// 语句的顺序不重要
  begin                    // p0_entry = p0; p1_entry =   p1;
    p0  <= ~i0 & ~i1; // p0_exit = ~i0 & ~i1;
    p1  <=i0 & i1;     // p1_exit = i0 & i1
    eq  <=p0 | p1;      //eq_exit = p0_entry | p1_entry
  end                      // eq = eq_exit; p0 = p0_exit; p1 = p1_exit;
endmodule
```

注意 p0 和 p1 均在敏感列表中。当 i0 或 i1 变化时，always 块被激活，新值在第一个时间步长的结束被赋值给 p0 和 p1。由于 eq 基于 p1 和 p1 的旧值(即，$p0_{entry}$ and $p1_{entry}$)，因此 eq 保持不变。当当前时间步长执行结束后，always 块因为 p0 和 p1 的变化(由于 p0 和 p1 在敏感列表中)再次被激活。eq 变量在第二个时间步长的最后使用 p0 和 p1 更新。注意，改变这些语句的顺序，结果是一样的。

虽然这两种代码描述的是同一个电路，但使用非阻塞赋值会消耗更多的仿真时间。因此，推荐使用阻塞赋值来描述组合逻辑。

7.1.3　存储元件

在 4.2 节中的存储元件模版中，非阻塞赋值被用于推断存储器。例如，一个 D 触发器的代码:

```
always@ ( posedge clk )
    q  <=d;
```

使用阻塞赋值来推断出一个存储器也是可能的。例如:

```
always@ (posedge clk)
    q = d;
```

虽然对于一个单独的触发器，上面的代码能正常工作。但多个寄存器相互作用时，就会导致一些微妙的问题出现。

考虑两个寄存器每个时钟都交换数据。当使用阻塞赋值，代码变为

```
always@ ( posedge clk )
    a = b;
always@ ( posedge clk )
    b = a;
```

在 clk 的上升沿，两个 always 块同时被激活和运行。这两个操作应在一个时间步长内完成。根据 Verilog 标准，两个 always 块的执行可以按任意顺序调度。如果先执行第一个 always 块，由于阻塞赋值，*a* 变量会马上得到 *b* 变量的值。当

第二个 always 块执行时，b 变量得到了 a 变量更新后的值，即 b 的原值，因此 b 变量不变。同理，如果第二个 always 块先被执行，a 得到其原值。这就是 Verilog 中的"竞态条件"。如果单从 Verilog 语言的角度看，这两种值都是合法的。

现在，我们使用非阻塞赋值修改上述代码(由于需要注释，因此增加了 begin 和 end 分隔符)：

```
always@ ( posedge clk )
begin          // b_entry = b
  a <= b;      // a_exit = b_entry
end            // a = a_exit
always@ ( posedge clk )
begin          // a_entry = a
  b <= a;      // b_exit = a_entry
end            // b = b_exit
```

非阻塞赋值解释详见注释。由于在赋值过程中使用原始的入口值，因此无论执行顺序如何，a 和 b 均会得到正确的值。

由于非阻塞赋值是对期望的行为建模，且避免了竞争条件，4.2 节所示的模板一直使用非阻塞赋值来推断触发器和寄存器。

7.1.4　时序电路使用阻塞和非阻塞赋值

在 4.2 节讨论的存储元件的模版都是最简单的时序电路代码。可以将多个赋值，包括阻塞和非阻塞赋值放于同一个 always 块中。我们用一个简单的例子解释不同组合的行为，近而更好地理解这两种赋值。

思考图 7-1b 所示电路。它将 a 和 b 进行与操作后，在时钟上升沿将结果存入一个 D 触发器中。基于我们之前的方法，我们可以将存储和组合电逻辑电路分离，得到两段代码，如示例 7.5 所示。

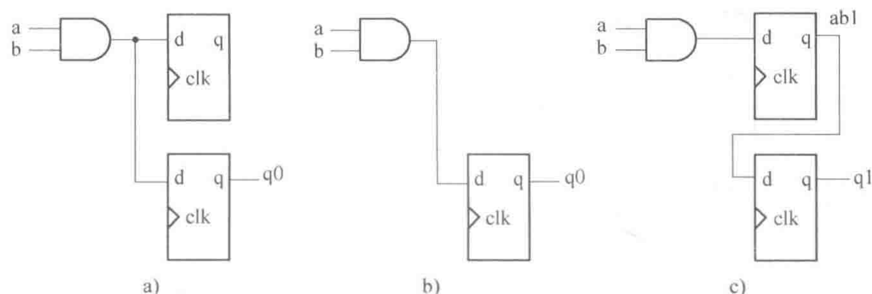


图 7-1　混合赋值推断出的电路

示例 7.5　两段式实现

```
module ab_ff_2seg
(
  input wire clk,
  input wire a, b,
  output reg q
);
  reg q_next;
  //D 触发器
  always @ ( posedge clk)
    q <= q_next;
  // 组合逻辑
  always @ *
    q_next = a & b;
endmodule
```

另外，我们可能将两段合并在一个 always 块中描述电路。示例 7.6 列出了阻塞和非阻塞的 6 种不同组合赋值代码。

示例 7.6　混合分配例子

```
module ab_ff _all
(
    input wire clk,
    input wire a, b,
    output reg q0, q1, q2, q3, q4, q5
);
    reg ab0, ab1, ab2, ab3, ab4, ab5;
    // attempt 0
    always @ (posedge clk)
begin
    ab0 = a & b;
    q0 <= ab0;
    end
// attempt 1
```

```
always @ (posedge clk)
begin                  // ab1_entry, = ab1; q1_entry = q1;
    ab1 <= a & b;      // ab1_exit = a & b
    q1 <= ab1;         // q1_exit = ab1_entry
end                    // ab1 = ab1_exit; q1 = q1_exit
//attempt 2
always @ (posedge clk)
begin
    ab2 = a & b;
    q2 = ab2;
end
//attempt 3 (交换attempt0 语句顺序)
always @ (posedge clk)
begin
    q3 <= ab3;
    ab3 = a & b;
end
//attempt 4 (交换attempt1 语句顺序)
always @ (posedge clk)
begin                  // ab4_entry, = ab4; q4_entry = q4;
    q4 <= ab4;         // q4_exit = ab4
    ab4 <= a & b;      // ab4_exit = a & b
end                    // ab4 = ab4_exit; q4 = q4_exit
//attempt 5 (交换 attempt 2 语句顺序)
always @ (posedge clk)
begin
    q5 = ab5;
    ab5 = a & b;
  end
endmodule
```

在 attemp 0 中，ab0 和 q0 的赋值最初推断出了两个寄存器，一个用于寄存 *ab*0，另一个用于寄存 *q*0。由于 *ab*0 通过阻塞赋值立即被更新，*q*0 得到 *a* & *b* 的值。相应电路电路图如图 7-1a 所示。由于 *ab*0 没有在 always 块外使用，对于 *ab*0 的寄存便没有必要，因此相应的寄存器便可以被移除。最终，电路如图 7-1b 所

示，这正是我们期望的电路。

在 attemp 1 中，对 *ab*1 的赋值使用了非阻塞赋值。相应解释见注释。注意，*q*1 得到 $ab1_{entry}$，而非 $ab1_{exit}$。$ab1_{entry}$，是 *ab*1 之前存储的值和相应的寄存输出。相应的电路如图 7-1c 所示。推断出了一个非预期的输入 buffer，*a* & *b* 的存储值被延迟了一个时钟周期。

在 attemp 2 中，阻塞赋值被应用到 *ab*2 和 *q*2。推断出的电路与 attemp 0 相同，如图 7-la 和图 7-1b 所示。由于使用阻塞赋值来推断触发器可能导致 7. 1. 3 节讨论的竞争条件，因此，这种代码风格是不推荐的。

为了便于示范，让我们检验当转换 attempts0、1、2 的赋值顺序会发生什么。转换后的代码如 attempts 3、4 和 5 所示。在 attemp3 中，*ab*3 在对其更新赋值之前被使用，这样，*q*3 得到了上一次激活时的旧值。这个值被存储于寄存器中，也就是寄存的 *a* &*b*。推断的电路对应图 7-1c。在 attempt4 中，切换顺序对代码功能没有影响，具体解释详见代码中的注释。它与 attemp1 相同。在 attemp5 中，*ab*5 在赋予新值之前被使用，这样，*q*5 得到了 *a* & *b* 寄存后的值。推断出的电路与 attempt3 相同。

总之，只有 attemp0 中的代码正确和可靠地描述了期望的电路。

7.2 另外一种时序电路代码风格

我们的时序代码模版遵循了图 4-2 所示的块图，它将寄存器分为一个单独的代码片段。当理解了阻塞和非阻塞赋值后，我们可以将寄存器和“next - state”逻辑放到同一个 always 块中。这种代码风格更紧凑。代码应遵循 7. 1. 4 节的 attemp1 片段：

- 使用阻塞赋值获取“next - state”逻辑的中间结果，这些赋值语句应以正确的顺序安排；
- 使用非阻塞赋值将中间结果赋值给寄存器。

在下面几节，我们用几个例子来说明这种风格。

7.2.1 二进制计数器

自由运行的计数器已经在 4. 3. 2 节讨论过了。我们可以修改示例 4. 9 的代码，将 next - state 逻辑和寄存器合并在一起，如示例 7. 7 所示。

示例 7. 7　将次态逻辑和寄存器合并的自由运行计数器

```
module bin_counter_merge
  #(parameter N=8)
```

```
  (
    input wire clk, reset,
    output wire  max_tick,
    output wire[N-1:0] q
);
// 信号声明
reg[N-1:0] r_next, r_reg;
// 程序体
// 寄存器和"next_state"逻辑
always @ ( posedge clk, posedge reset)
    if(reset)
        r_reg <=0; // {N {1b'0}}
    else
        begin
            // 状态逻辑
            r_next = r_reg +1;
            // 寄存器
            r_reg <=r_next;
        end
    // 输出逻辑
    assign q =r_reg;
    assign max_tick =(r_reg = =2 * * N -1) ? 1'b1: 1'b0;
endmodule
```

注意输出逻辑的描述：

```
assign max_tick =(r_reg = =2 * * N -1) ? 1'b1: 1'b0;
```

必须在always块外，如果放在块内，会为max_tick推断出一个额外的触发器，并引入一个时钟周期的延时。

由于r_next没有在其他场合使用，我们可以将两个语句合并：

```
r_next =r_reg +1;
r_reg <=r_next;
```

变为

```
r_reg <=r_reg +1;
```

用q代替r_reg后，代码会更加简洁，如示例7.8所示。

示例 7.8　使用紧凑代码实现的自由运行二进制计数器

```
module bin_counter_terse
  # ( parameter N = 8)
  (
  input wire clk, reset,
  output wire max - tick,
  output reg [N - 1: 0] q
  );
  // 程序体
  always @ (posedge clk, posedge reset)
    if ( reset )
        q <= 0;
    else
        q <= q + 1;
    // 输出逻辑
  assign max_tick = (q = = 2 * * N - 1) ? 1'b1 : 1'b0;
endmodule
```

在这段代码中，等式右边的 q 是寄存器的输出，等式左边的 q 是更新值，它在下一个时钟上升沿存储到寄存器中。

示例 4.10 中列出的通用二进制计数器可以用同样的方法修改，代码如示例 7.9 所示。

示例 7.9　合并寄存器和次态逻辑的通用二进制计数器

```
module univ_bin_counter_merged
  # (parameter N = 8)
  (
    input wire clk, reset,
    input wire syn_clr, load, en, up,
    input wire [N - 1: 0] d,
    output wire max_tick, min_tick,
    output reg [N - 1: 0] q
  );
  // 程序体
  // 寄存器和次态逻辑
```

```
always @ ( posedge clk, posedge reset)
  if (reset)
    q <=0; //
  else if (syn_clr)
    q <=0;
  else if (load)
    q <=d;
  else if (en & up)
    q <=q +1;
  else if (en & ~up)
    q <=q - 1;
  // 由于q <=q 隐式实现，因此无其他分支
// 输出逻辑
assign max_tick =(q = =2 * *N -1) ? 1'b1 : 1'b0;
assign min_tick =(q = =0) ? 1'b1: 1'b0;
endmodule
```

注意最后的 else 分支被省略掉了，它暗含 q 保持其之前的值，即这正是期望的行为。

7.2.2 FSM

有限状态机中的状态寄存器和次态逻辑也可以按类似的方法合并。例如，思考示例 5.1 中的状态机。修改后的代码如示例 7.10 所示。

示例 7.10　寄存器与次态逻辑合并的有限状态机

```
module fsm_eg_merged
  (
    input wire clk, reset,
    input wire  a, b,
    output wire y0, y1
  );
  // 状态符号声明
  parameter [1: 0] s0 =2'b00,
                   s1 =2'b01,
                   s2 =2'b10;
```

```
  // 信号声明
  reg [1: 0] state_reg;
  // 状态寄存器和次态逻辑
  always @ (posedge clk, posedge reset)
      if (reset)
        state_reg <= s0;
      else
        case (state_reg)
          s0: if (a)
                 if (b)
                     state_reg <= s2;
                 else
                     state_reg <= s1;
               else
                 state_reg <= s0;
          s1: if (a)
                 state_reg <= s0;
               else
                 state_reg <= s1;
          s2: state_reg <= s0;
          default: state_reg <= s0;
        endcase
  // Moore 输出逻辑
  assign y1 = (state_reg == s0) || (state_reg == s1);
  // Mealy 输出逻辑
  assign y0 = (state_reg == s0) & a & b;
endmodule
```

由于输出未寄存，因此，相应语句必须置于 always 块外。

7.2.3 FSMD

该方法也可应用于 FSMD。思考示例 6.5 中的除法 FSMD 例子。修改后的代码见示例 7.11。

示例 7.11　状态寄存器和组合逻辑合并的除法 FSMD

```
module div_combined
```

```
#(
  parameter W = 8,
            CBIT = 4  //CBIT = log2 (W ) + 1
  )
(
  input wire clk, reset,
  input wire start,
  input wire [W - 1: 0]dvsr, dvnd,
  output wire ready, done_tick,
  output wire [W - 1: 0] quo, rmd
);
//状态符号声明
localparam [1: 0]
    idle = 2'b00,
    op   = 2'b01,
    last = 2'b10,
    done = 2'b11;
//信号声明
reg [1: 0] state_reg;
reg [W - 1: 0] rh_reg, rl_reg, rh_tmp, d_reg;
reg [CBIT - 1: 0] n_reg, n_next;
reg q_bit;
//fsmd 寄存器和次态逻辑
always @ (posedge clk, posedge reset)
begin
    if (reset)
      begin
          state_reg <= idle;
          rh_reg <=0;
          rl_reg <=0;
          d_reg <=0;
          n_reg <=0;
      end
    else
begin
```

```
// ==================================================
// 数据路径功能单元
// 用于获取中间结果
// ==================================================
// 比较和减法电路
if (rh_reg >=d_reg)
  begin
      rh_tmp = rh_reg - d_reg;
      q_bit = 1'b1;
  end
else
  begin
      rh_tmp = rh_reg;
      q_bit = 1'b0;
  end
// 索引值递减
n_next = n_reg - 1;
// ==================================================
// 状态、数据寄存器和次态逻辑
// ==================================================
case (state_reg)
  idle:
      begin
        if (start)
            begin
              rh_reg <=0;
              rl_reg <=dvnd;    // 被除数
              d_reg <=dvsr;     // 除数
              n_reg <=CBIT;     // 索引值
              state_reg <=op;
            end
      end
  op:
      begin
              //rh 和rl 左移
```

```
                rl_reg  <= {rl_reg[W-2: 0], q_bit};
                rh_reg  <= {rh_tmp[W-2: 0], rl_reg[W-1]};
                // 减小索引值
                n_reg  <= n_next;
                if (n_next == 1)
                    state_reg  <= last;
            end
         last: // 最后一次重复
            begin
                rl_reg  <= {rl_reg[W-2: 0], q_bit};
                rh_reg  <= rh_tmp;
                state_reg  <= done;
            end
         done:
            state_reg  <= idle;
         default: state_reg  <= idle;
      endcase
    end
  end
  // 输出
  assign quo = rl_reg;
  assign rmd = rh_reg;
  // 非寄存输出
  assign ready = (state_reg == idle);
  assign done_tick = (state_reg == done);
endmodule
```

这段代码更加复杂，并包含了一段数据路径功能的单元，用于产生中间结果。注意，一些中间变量，如 n_next，被用于后面的许多地方。

7.2.4　总结

总之，将次态逻辑和寄存器放于一段 always 块中是可能的。这种代码风格更紧凑，使用的变量更少。然而，必须小心编码，避免产生其他多余的寄存器。

7.3 使用有符号数据类型

7.3.1 概述

根据应用的特点，可以在数字系统中使用由 0 和正数组成的无符号整数，或者由 0 以及正数和负数组成的有符号整数。但在一个复杂的系统中，可能同时需要这两种数据类型。

有符号整数通常由二进制补码格式表示。图 7-2 所示为一个 4 位的“二进制码盘”，里面罗列了二进制码以及对应的无符号和有符号字。仔细观察会发现这两类数字的加法和减法操作是相同的，加减一个正数的结果分别对应着向顺时针和逆时针的移动。例如，“1001” + “0100”意味着从“1001”位置顺时针移动 4 个位置，结果便是“1101”。在无符号整型格式中，翻译为(+9) + (+4) = +13，在有符号整型格式中，翻译为(-7) + (+4) = -3。加法的溢出对应着移动过码盘的“阈值”。注意，无符号和有符号数的阈值是不一样的。无符号数的阈值在“1 11 1” 和“0000”之间，而有符号数的阈值在“01 1 1”和“1000”之间。

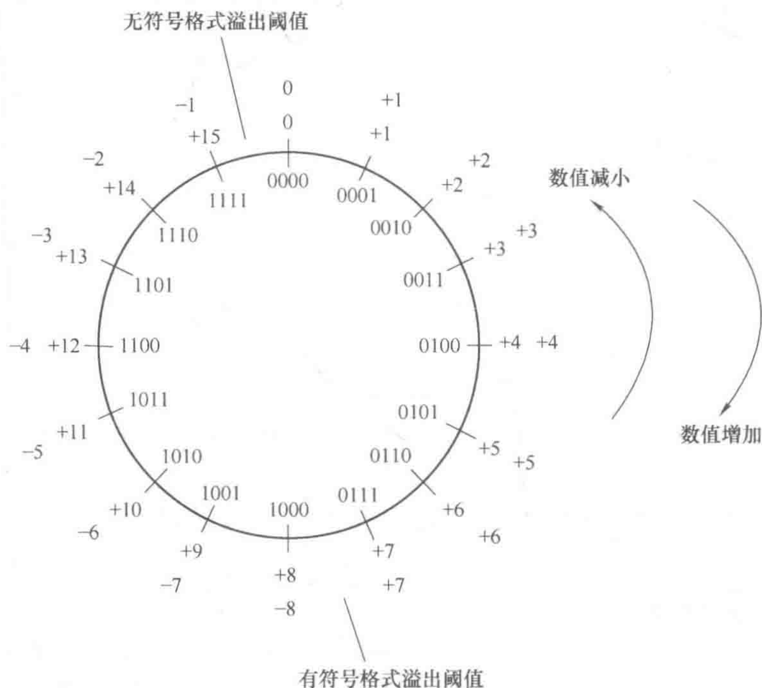


图 7-2　4bit 二进制码盘

物理上的加减法行为就像是二进制码盘的移动。同样的电路可适用于有符号和无符号格式的运算，只要所有的操作数和结果的比特位数相同。例如，设 a、b 和 sum 为 3 个 8bit 信号。语句：

sum = a + b;

便会推断出同样的硬件电路，并使用同样的二进制表示，无论信号是否被当成无符号数还是有符号数。这一定理在其他算法操作中同样是正确的(遗憾的是，无法用于非算法操作中，如关系操作或溢出状态产生)。

另外，当操作数或结果的位宽不同时，需要区分格式。这是因为在位宽扩展上需求不同。对于无符号格式，高位填充 0，即“0 扩展”，但对于有符号格式，高位扩展的是符号位，即“符号扩展”。例如，－5 由 4bit 表示为“1011”，当扩展到 8bit 时，变为“1111－1011”，而非“0000－1011”。

例如，设 a 和 sum 为 8 位有符号数，b 为 4 位有符号数，$b_3b_2b_1b_0$，语句：

sum = a + b;

需要把 b 扩展到 8 位。在无符号格式中，扩展后的 b 为 $0000b_3b_2b_1b_0$，而在有符号格式中，扩展后的 b 为 $b_3b_3b_3b_3b_3b_2b_1b_0$。语句所推断出的硬件包含位扩展电路和一个加法器。由于扩展电路对于有符号和无符号格式有所有同，因此，对于有符号和无符号格式，语句会推荐出不同的电路实现。

7.3.2 Verilog－1995 中的有符号数

在 Verilog－1995 中，只有 integer 数据类型可以被当作有符号数，reg 和 wire 数据类型会被当作无符号数字。由于 integer 数据类型位宽固定(通常是 32 位)，因此使用上并不灵活。为了实现有符号操作，常常需要手动编码。如下的代码片段说明了有符号和无符号操作。

```
reg[7:0] a, b;
reg[3:0] c;
reg[7:0] sum1, sum2, sum3, sum4;
// 相同位宽，可用于有符号和无符号操作
sum1 = a + b;
// 自动扩展0
sum2 = a + c;
// 手动扩展0
sum3 = a + {4{1'b0}, c};
// 手动符号扩展
sum4 = a + {4{c[3]}, c};
```

第一句中的 a、b 和 sum1 位宽相同，这样，无论它们被当作有符号数还是无

符号数，推断出的电路是一样的。

第二句中的 c 位宽只有 4bit。要根据 3.2.8 节讨论的规则调整它的位宽。由于 reg 类型是能被用作无符号数，因此会采用 0 扩展，在 c 的前面添加 4 个 0。

在第三句中，我们手动将 4 个 0 添加到 c 的前面，达到了与前面语句相同的效果。

在第四句中，我们将变量当作有符号数。为了实现期望的行为，必须通过符号扩展将 c 扩展到 8 位。只能通过人手编码完成，将 c 的 MSB 复制 4 次（即 4{c[3]}）来生成符号扩展的 8bit 数。

7.3.3 Verilog-2001 中的有符号数

在 Verilog-2001 中，reg 和 wire 数据类型可以通过在声明中使用 signed 关键字扩展为有符号格式，如下所示：

```
reg signed [7:0] a, b;
```

使用有符号数据类型，之前的代码片段可修改为

```
reg signed [7:0] a, b;
reg signed [3:0] c;
reg signed [7:0] sum1, sum4;
...
// 相同位宽，可用于有符号和无符号操作
sum1 = a + b;
// 自动符号扩展
sum4 = a + c;
```

由于 a、b 和 sum1 位宽相等，因此第一句推断出一个常规的加法器，signed 数据类型只是帮助我们要关注对于二进制表示的翻译。

第二句中表达式左侧的所有变量都是 signed 数据类型，且 c 会通过符号扩展自动扩展至 8 位。这样，我们便无需对变量进行手动填充。

在小规模数字系统中，通常只使用无符号或有符号格式中的一种。然而，在较大的系统设计中，可能包含不同格式的子系统。Verilog 是一种类型比较宽松的语言，无符号和有符号变量可以在同一表达式中混合使用。根据 Verilog 标准，只有当表达式右边的所有变量均为有符号数，才会进行符号位扩展。否则，所有的变量都会进行 0 扩展。思考如下代码片段：

```
reg signed [7:0] a, sum;
reg signed [3:0]b;
reg [3:0] c;
...
```

sum = a + b + c;

由于c没有使用signed数据类型，因此，表达式右侧的变量b和c都是0扩展。

Verilog提供了两个系统函数，$signed( )和$unsigned( )，用于将括号内的表示转换为signed和unsigned数据类型。例如，可以用下面的语句转换c的数据类型：

```
sum = a + b + $signed(c);
```

现在，表达式右侧所有3个变量均为signed数据类型，这样b和c就是符号扩展了。

将signed和unsigned数据类型混合在一个复杂的表达式中易引入一些微妙的错误，因此应避免这种使用。如果真是必须要这样做，那么表达式应保持简洁，且应使用转换函数，以确保数据类型的一致性。

7.4 在综合中使用函数

7.4.1 概述

在Verilog的module中，一些表达式可能出现在许多地方。共同使用的部分应当被提取到一个例行程序中，而不是重复进行编码。可以通过在module中定义一个函数实现。Verilog的函数可以带有一个或多个输入参数，并返回一个单一值。在综合过程中，函数被展开和"打散"，并映射到硬件中。这样，为了综合起见，对于复杂的表达式，函数应保持简明，并可作为速记使用。函数的基本语法为

```
module
...
// 函数定义于module 内
function [result_type] [func_id] ([input_arg]);
  begin
    [statements];
  end
endfunction
...
endmodule
```

函数体定义在function和endfunction分隔符内。可选项[result-type]用于指定返回结果的数据类型，通常是带有位宽定义的reg或integer。输入参数声明在[input-arg]中，函数的名字通过[func-id]指定。函数体通过语句描述，结果通过

如下的语句返回：

[func_id] = …;

7.4.2 举例

回顾示例 6.6 所示的二进制到 BCD 码的转换电路。在转换过程中，需要对每个 BCD 数字用特定的方式加一个值。为了让 FSMD 部分代码更加清楚，我们在代码中用分离的片段说明：

```
module...
...
assign  bcd0_tmp = (bcd0_reg > 4) ? bcd0_reg + 3 : bcd0_reg;
assign  bcd1_tmp = (bcd1_reg > 4) ? bcd1_reg + 3 : bcd1_reg;
assign  bcd2_tmp = (bcd2_reg > 4) ? bcd2_reg + 3 : bcd2_reg;
assign  bcd3_tmp = (bcd3_reg > 4) ? bcd3_reg + 3 : bcd3_reg;
...
endmodule
```

我们定义一个函数 ba()来代替重复编码四次同样的表达式来实现相同的目的。修改后的代码片段成为了：

```
module...
assign bcd0_tmp = ba(bcd0_reg);
assign bcd1_tmp = ba(bcd1_reg);
assign bcd2_tmp = ba(bcd2_reg);
assign bcd3_tmp = ba(bcd3_reg);
...
// 函数定义(ba : bcd adjust)
function [3:1] ba ( reg [3: 0] bcd_in);
begin
  ba = (bcd_in > 4) ? bcd_in + 3 : bcd_in;
end
endfunction
...
endmodule
```

函数 ba()(用于 BCD 转换)在最后被定义。函数使用了一个 4bit 的参数，并返回一个 4bit 的结果。可以使用此函数替换之前的表达式。事实上，可以使用函数 bc(bcd_reg)直接代替 bcd0_tmp，将这些变量从代码中去除。

另外一个常用的函数是用于计算依赖其他参数的常数。回顾在示例 4.11 中

讨论的模 m 计数器。有两个参数：用于指定 m 值的 M，以及用于指定计数器中位宽的参数 N。N 的值为$\lceil \log_2 M \rceil$，且不应是一个独立的参数。一个更好的方法是指定 N 作为一个局部参数，并在 module 内部计算它的值。可以通过函数实现。修改后的代码如示例 7.12 所示。

示例 7.12　使用函数的模 m 计数器

```
module mod_m_counter_fc
  #(parameter M = 10) //mod-M
  (
    input wire clk, reset,
    output wire max_tick,
    output wire [log2(M) -1: 0] q
  );
  // 信号声明
localparam N = log2(M);    //M 值的位宽
  reg [N -1: 0] r_reg;
  wire [N -1: 0] r_next;
  // 程序体
  // 寄存器
  always @ (posedge clk, posedge reset)
      if (reset)
        r_reg  <=0;
      else
        r_reg  <= r_next;
  // 次态逻辑
  assign r_next = (r_reg = = (M -1)) ? 0 : r_reg + 1;
  // 输出逻辑
  assign q = r_reg;
  assign max_tick = (r_reg = = (M -1)) ? 1'b1 : 1'b0;
  //log2 函数
  function integer log2(input integer n);
      integer i;
  begin
      log2 = 1;
      for (i =0; 2 * * i  < n; i = i +1)
```

```
        log2 = i + 1;
  end
  endfunction
endmodule
```

可计算[$\log_2(x)$]的函数 log2( )被定义在 module 内，用于获取局部参数 N，由于运算是在代码被细化时执行的，因此在综合前数值就已经确定了，且不会有任何物理电路被该函数推断出。

7.5　用于测试平台开发的额外结构

由于本书主要关注于硬件的开发，因此我们只对一小部分可综合的 Verilog 子集进行了检验，在验证上，也只使用了两种基本的测试平台模板。虽然详细覆盖 Verilog 语言和测试平台的内容已经超出了本书的要求，但在这一节，会对一些语言结构提供一个简明的概述，以帮助我们开发更加复杂的测试平台。

不像可综合代码，测试平台代码会被送到仿真器中，并在一个主机中执行。可以在代码中包含复杂的语言结构和顺序执行的算法。许多 Verilog 结构与 C 语言中的很相似，并可以用相似的方式使用。

7.5.1　always 和 initial 块

Verilog 有两种类型的过程性代码块：always 和 initial。一个 always 块里包含了程序语句并建模了一个抽象电路部件。我们已经在 3.3 节中对一种特定类型的 always 块进行了检验。它是用于综合的。块里有敏感列表，但没有包含其他明显的时序控制结构。always 块的激活和执行通过敏感列表指定的事件触发。

若出于建模的目的，always 块便可以包含时序结构，以指定不同结构的相应的延时或等待特定的事件。有时敏感列表可以省略。例如，可以使用下列代码片段对时钟信号建模，第 20 个时间单元：

```
always
begin
   clk = 1'b1;
   #20;
   clk = 1'b0;
   #20;
end
```

initial 块内也包含程序语句。但它只在仿真的一开始执行，且只执行一次。

简化的语法为

```
initial
begin
    [procedural statements]
end
```

initial 块通常用作设置变量的初值，在示例1.7中，被用作生成整个测试序列。initial 块“运行一次”的行为通常是不可综合的。

7.5.2 程序语句

程序语句在 initial 块、always 块、function 和 task 内使用，经常使用的程序语句有：

- 阻塞赋值；
- 非阻塞赋值；
- If 语句；
- case 语句；
- 循环语句。

我们已经在7.1节讨论了阻塞和非阻塞赋值。并在3.4节和3.5节讨论了了 if 和 case 语句。

Verilog 支持4种循环结构：for、while、repeat 和 forever。for 循环的简化语法为

```
for([initial-assignment]; [end - condition]; [step-assignment])
begin
  [procedural-statements;]
end
```

例如，可以清除16word 的寄存器组。

```
integer i;
for (i =0; i <16; i =i +1)
    reg_ file [i] =0;
```

注意，如果只有一句语句，begin 和 end 分隔符可以省略。while 循环的简化语法为

```
while ([end-condition])
begin
    [procedural-statements;]
end
```

loop 循环体内的语句不停地重复，直到[end-condition]条件表达式满足。例

如，之前的清除寄存器操作也可以用 while 循环完成。

```
integer i;
...
i =0;
while (i <16)
begin
  reg_file [i] =0;
  i =i +1;
end
```

repeat 循环的简化语法为

```
repeat([number])
begin
  [procedural-statements;]
end
```

循环体内的循环会重复特定次数，通过指定[number]参数实现。例如，之前的操作也可以用 repeat 循环完成：

```
integer i;
...
i =0;
repeat (16)
begin
    reg-file [i] =0;
    i =i +1;
end
```

forever 循环的简化语法为

```
forever
begin
  [procedural-statements;]
end
```

forever 循环，顾名思义，就是重复执行循环体，直至仿真结束。循环体通常包含特定的时序控制结构，这样可以被系统周期性挂起。例如，下面的代码片段是另外一种描述时钟信号的方法，每隔 10 个时间单元翻转，并一直运行。

```
initial begin
clk =1'b0;
forever
```

```
  #10 clk = ~clk;
end
```

7.5.3 时序控制

在测评平台中，必须指定不同信号的激活和无效时间，或者等待某些事件或条件。有三种时序控制结构：

- 延时控制：# [delay-time]；
- 事件控制：@ ([eventl, [eventl, ...)；
- 等待控制：wait ([boolean_expression])。

另外，一个编译指令，'timescale，也与时序规格有关。

7.5.4 延时控制

延时控制通过#符号与紧跟着的时间值表示。通过指定的数目来延迟程序语句的执行。若延时控制放置于左边，则整个语句的执行就会被延迟。例如，思考下列代码片段：

```
...
#10 a = 1'b0;
#5 y = a | b;
...
```

假设当前的仿真时间为 t，上述语句表示 a 将会在 $t+10$ 时刻获取 0，再过 5 个单位时间(即在 $t+15$ 时刻)，a | b 表达式的值被计算并将结构赋值给 y。

如果延时控制被放置在右边，表达式的计算会立即执行，但赋值给左边变量的行为会被延迟。思考下面代码片段：

```
...
#10 a = 1'b0;
y = #5 a | b;
...
```

同样，a 会在 $t+10$ 时刻获取 0，a | b 表达式的求值会马上进行(即在 $t+10$ 时刻)，但在 $t+15$ 时刻才会将结果赋值给 y。

经常会使用延时控制在测试平台中产生一个激励，而不是用于建模传播延时。下面格式的代码更直观些：

```
...
a = 1'b0; //a 为0
#10; //0 值持续10 个时间单位
a = 1'b1; //a 变为1
```

```
#5//1 值持续5 个时间单位
a =1' b0; //a 变为0
#20//0 值持续20 个时间单位
...
```

7.5.5 事件控制

事件控制通过@符号与紧跟着的敏感列表表示，敏感列表可指定期望的事件。事件控制与在 always 块中的使用相似。事件是指敏感列表信号的值发生了变化(即信号转换)。posedge 和 negedge 两个关键字可以添加到敏感列表中，以指定期望的转换沿(即上升沿或下降沿)。在测试平台中，执行会被挂起，直到其中一个指定的事件发生。事件控制的一个常见应用是将激励的产生与时钟信号同步。例如，下面的代码片段置使能信号 en 有效一个时钟周期。

```
localparam delta =1;
...
@ (posedge clk) // 等待时钟上升沿
#delta; // 等待delta 时间，避免保持时间违反
en =1'b1 // 置en =1 有效
@ (posege clk) // 等待下一个时钟上升沿
#delta; // 等待delta 时间，避免保持时间违反
en =1'b0 // 置en =0 无效
...
```

另外，也可以在时钟下降沿置 en 信号有效和无效：

```
...
@ (negdge clk) // 等待时钟下降沿
en =1'b1 // 置en =1 有效
@ (negdge clk) // 等待下一个时钟下降沿
en =1'b0 // 置en =0 无效
```

7.5.6 wait 语句

wait 语句等待某个特定的条件，简化的语法为

wait [boolean_expression]

后面语句的执行将会被挂起，直到通过[boolean-expression]指定的条件为真。例如，可以编码这样的代码：

wait (state = READ && men_ready = =1 'b1) [statement_to_get_data];

也可以使用 wait 语句挂起执行。例如，可以等待一个计数器计到 15，然后

再激活某个信号：

```
...
wait(counter = =4'b1111); // 等待，直到计数15 次
...                        // 继续
```

wait语句与事件控制有些相似。后者等待特定信号的转换沿，而前者等待特定的条件，有时被认为是电平敏感。

7.5.7 timescale指令

编译器指令被用于控制Verilog代码的编译和处理。前面标有重音符（'），通常位于键盘的左上角。其中一个时序相关指令是'timescale指令，它的语法为

'timescale [time-unit] / [time-precision]

[time-unit]用于指定时间和延时测试单元，[time-precision]用于指定仿真的分辨率。

例如，指令：

'timescale 10ns / 1ns

表示仿真的单位是10ns，分辨率为1ns。当延时在代码中指定时，如：

#5 y = a & b;

这表示实际的延时是50ns（即5×10ns）。

延时规格可为单位的小数，如：

#5.12345 y = a & b;

表示实际延时是51.2345ns，由于精度是1ns，因此会在仿真中被舍入到51ns。更高的精度会增加仿真的准确性，但可能会减小仿真速度。

[time-unit]和[time-precision]数字部分可以为1、10或者100，时间单位可为s（秒）、ms（毫秒）、μs（微秒）、ns（纳秒）、ps（皮秒）或者fs（飞秒）。

7.5.8 系统函数和任务

Verilog有一组定义好的系统函数和任务，执行系统相关的操作，例如仿真控制和文件存取。这些函数和任务以美元符$开头。我们将会在这一小节检验一些常用的函数和任务。

1. 数据类型转换函数 $ unsigned和$ signed函数用于无符号和有符号数据类型之间的转换。它们的使用已经在7.3节讨论过。

2. 仿真时间函数 仿真时间函数返回当前的仿真时间，$ time、$ stime和$ realtime三个函数分别将时间按64位、32位和实数类型返回。

3. 仿真控制任务 有两个仿真控制任务，$ finish和$ stop。$ finish用于结束仿真并退出仿真程序。$ stop用于挂起仿真。在ModelSim中，以交互的方

式返回仿真。在开发流程中，经常会停留在 ModelSim 环境中来做近一步编辑或检验波形，可以在代码中使用 $ stop 任务完成。

4. 显示任务 2.4 节讨论的开发流程与在实验平台做实验很相似。仿真结果以波形形式在 ModelSim 中显示。ModelSim 模仿实验平台上使用的逻辑分析仪。另外一种显示结果的方式是文本。4 个主要的显示系统函数为 $ display、$ wirte、$ strobe 和 $ monitor。这几个函数有着相似的语法，可以在仿真过程中显示文本。在 ModelSim 中，文本显示在控制台上。

$ display 的格式与 C 语言的打印函数相似。简化的语法为

$ display ([format_string], [argument], [argument], ...);

[format - string]包含了常规的字符和“换码顺序”，用以指定相应参数的显示格式。当字符串被显示时，相应参数值会被代替到字符串中，并以指定的格式显示。例如，在下列语句中：

$ display ("at %d; signal x = %b", $ time, x);

%d 和%b 是换码顺序，用于指定当前仿真时间和 x 要以十进制和二进制格式显示，相应地，结果显示为

at 5100; signal x = 00110001

仿真中常用的换码顺序有%d、%b、%o、%h、%c、%s 和%g，分别表示十进制、二进制、八进制、十六进制、字符、字符串和实数。

$ write 任务几乎与 $ display 任务相同，唯一的区别就是 $ write 不会在字符的末尾添加换行符。相关显示任务从当前位置继续输出。换行符，\n，必须手动添加到字符串中来结束一行的显示。

Verilog 包含了时间片的概念，用来对传播延迟进行建模。如 7.5.7 节所讨论的。许多活动可以在一个时间片内发生。$ strobe 任务与 $ display 任务相似。$ strobe 在当前的时间片的末尾执行，而不是立即执行。避免了由于竞态条件导致的数据显示不一致的问题。

$ monitor 任务是一个非常通用的命令，鉴于 $ display、$ write 或 $ strobe 只在每次执行时显示一次文本，$ monitor 在每次参数值发生变化时，都会显示。$ monitor 为仿真的跟踪提供了简单灵活的方式。例如，可以在示例 1.7 的测试平台中添加下列代码片段：

```
initial
begin
   $display("time   test_in0   test_in1   test_out");
   $monitor("%d        %b            %b             %b",
              $time, test_in0, test_in1, test_out);
end
```

文本仿真结果显示在控制台中：

time	test_in0	test_in1	test_out
0	00	00	1
200	01	00	0
400	01	11	0
600	10	10	1
800	10	00	0
1000	11	11	1
1200	11	01	0

文件 I/O 系统　函数与任务 Verilog 提供了一套用于访问数据文件的函数和任务。文件可通过 $ fopen 和 $ fclose 函数打开和关闭。$ fopen 的简化语法为

[mcd - name] = $ fopen ("[file - name]");

$ fopen 函数返回一个与文件相关的 32 位多通道描述符。描述符可被看作一个 32 位的标准，每一位代表一个文件(也就是通道)。最低位为标准输出(即控制台)保留。当函数被调用且文件成功打开，函数返回一个带有 1bit 有效位的描述符。例如，0…0010 是第一个打开的文件返回值，0......0100 是第二个打开的文件返回值，依次类推。函数返回值全为 0 表示打开失败。

一旦文件被打开，便可以使用 4 种改进的显示系统任务（$ fdisplay、$ fwrite、$ fstrobe 和 $ fmonitor）写入数据。除了多通道描述符作为第一个参数被使用，与原始的任务相似。如：

$ fdisplay([mcd_name], [format_string], …);

一个简单示例代码片段如示例 7.13 所示。

示例 7.13　写文件示例

```
integer log_file, both_file;
localparam con_file = 16'h0000_0001;                    // 控制句柄
initial
begin
    log_file = $ fopen("my_log");
    if (log_file = =0)
        $ display(" Fail to open log file"); // 写句柄
    // 向句柄和日志文件中写入
    $ fdisplay( both_file, " Simulation Started" );
    …
    // 只向日志文件中写入
```

```
    $ fdisplay(log_file, … );
    …
    // 向句柄和日志文件中写入
    $ fdisplay(both_file, " simulation ended" );
    $ fclose(log_file);
end
```

注意，可以通过对多通道描述符进行位运算或操作来创建新的描述符，如示例中的 both_file 变量。当 both_file 被使用时，文本就会被写入到控制台和日志文件中。

Verilog 获取外部文件数据有两个简单的系统任务：$ readmemb 和 $ readmemh。这两个任务假设外部文件存储内容分别是二进制和十六进制格式的数组。简化的语法为

```
$ readmemb("[file_name]", [mem_variable]);
$ readmemh("[file_name]", [mem_variable]);
```

下列代码片段对 8 ×4 数组的获取进行了说明：

```
reg [3: 0] v_mem [0: 7]
…
$ readmemb("vector. txt",    v_mem);
…
```

文件中应包含 8 个 4bit 二进制数据，并通过空格分隔。

使用文件操作函数和任务，将使用外部分件指定测试向量和记录仿真结果成为可能。回顾示例 1.7 的测试平台，可以修改为文件操作，如示例 7.14 所示。

示例 7.14　基于文件操作的测试平台

```
`timescale 1 ns/10 ps
module eq2_file_tb;
  // 信号声明
  reg  [1: 0] test_in0, test_in1;
  wire  test_out;
  integer log_file, console_file, out_file;
  reg [3: 0] v_mem [0: 7];
  integer i;
  // 实例化被测电路
  eq2_sop uut
```

```
    (.a(test_in0), .b(test_in1), .aeqb(test_out));
initial
begin
    //设置输出文件
    log_file = $fopen("eqlog.txt");
    if (! log_file)
       $display("Cannot open log file");
    console_file = 32'h0000_0001;
    out_file = log_file | console_file;
    //读测试向量
    $readmemb("vector.txt", v_mem);
    //测试发生器循环遍历8 个测试向量
    for(i=0; i<8; i=i+1)
      begin
        {test_in0, test_in1} =v_mem[i];
        #200;
      end
    //仿真停止
    $fclose(log_file);
    $stop;
end
//文本显示
initial
begin
  $fdisplay(out_file,"        time   test_in0   test_in1   test_out");
  $fdisplay(out_file,"                 (a)        (b)       (aeqb) ");
  $fmonitor(out_file,"%10d      %b          %b          %b",
                           $time, test_in0, test_in1, test_out);
end
endmodule
```

测试向量为存入于 vector.txt 文件中的指定的 4bit 二进制数据。文件内容为

```
00_00
01_00
01_11
```

10_10
10_00
11_11
11_01
00_10

文件被读入到二维变量 v_mem 中。测试向量生成器使用一个 for 循环遍历 8 个测试向量。仿真结果被写入到控制台和日志文件 eqlog. txt 中。日志文件的内容为

time	test_in0	test_in1	test_out
	(a)	(b)	(aeqb)
0	00	00	1
200	01	00	0
400	01	11	0
600	10	10	1
800	10	00	0
1000	11	11	1
1200	11	01	0
1400	00	10	0

日志文件是一个常规文本文件，可以被任何文本编辑器查看。

7.5.9 自定义函数和任务

一个复杂的测试平台可能会非常冗长和繁杂。一种控制这种复杂性的方式是将代码分割成一些更小的部分。函数和任务可以帮助实现这件事。我们在 7.4 节已经讨论了 Verilog 的函数。函数带有输入参数并返回一个数值。当被调用时，函数会马上被执行且函数内部不允许有时序控制结构。

任务更加灵活和通用。它可以有输入、输出和双向的参数，还可以包含时序控制结构。通过输出和双向参数，多个值可以被返回。与函数相同，任务也必须在模块内部声明。任务的基本语法为

```
task [task_id] ([arg]);
    begin
        [statements];
    end
endtask
```

[arg] 为参数声明部分。除了默认数据类型是 reg 以及 wire 类型不能被使用外，它的语法格式与端口声明相似。示例 7.15 展示了使用任务实现的 2bit 相等

比较器的建模。

示例 7.15 使用任务实现的 2bit 比较器

```
module eq2_task
  (
  input   wire [1: 0] a, b,
  output reg aeqb
  );
  reg e0, e1;
always @ *
  begin
    equ_tsk(2, a[0], b[0], e0);
    equ_tsk(2, a[1], b[1], e1);
    aeqb = e0 & e1;
  end
//task 定义
task equ_tsk
  (
    input integer delay,
    input i0, i1,
    output eq1
  );
  begin
      #delay eq1 = ( ~i0 & ~i1) | (i0 & i1);
  end
endtask
endmodule
```

注意，操作的传播延时通过#delay 指定，延时值通过 delay 参数传入任务中。出于比较目的，使用函数重写的代码，详见示例 7.16。

示例 7.16 使用函数实现的 2bit 比较器

```
module eq2_function
  (
  input   wire [1: 0] a, b,
```

```
output reg aeqb
  );
  reg e0, e1;
always @ *
  begin
    #2 e0 = equ_fnc(a[0], b[0]);
    #2 e1 = equ_fnc(a[1], b[1]);
    aeqb = e0 & e1;
end
// 功能定义
function equ_fnc(input i0, i1);
  begin
    equ_fnc = ( ~i0 & ~i1) | (i0 & i1);
  end
endfunction
endmodule
```

注意，函数不能包含时序控制。为达到同样的效果，延时必须只能在 always 块中指定。

7.5.10 复杂测试平台示例

学习完这些附加的语言结构，便可以开发一个更加复杂的测试平台。再次考虑示例 4.10 中的通用二进制计数器的测试平台。新的测试平台的概要框图如图 7-3 所示。里面有 3 个模块。除了计数器，bin_gen 模块生成测试向量，monitor 模块监视输入激励和输出响应。

测试向量生成模块　如示例 4.12，直接生成测试向量是个冗余和乏味的过程。一个更好的选择是开发一套对应不同操作的抽象程序。这会使代码更好组织更易理解。单独的程序可以使用任务完成。例如在前面的测试平台中，可以定义一个用于实现计数器数据载入操作的任务：

```
task load_data(input wire [N-1 : 0] data_in);
begin
    @(negedge clk);    // 等待下降沿
    load = 1'b1;
    d = data_in;
    @(negedge clk);
```

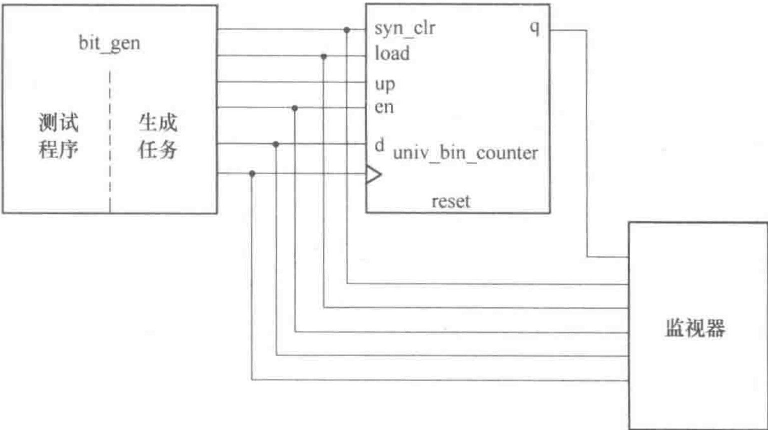


图 7-3　复杂的测试平台结构框图

```
      load = 1'b0;
   end
   endtask
```

在任务中，load 在两个下降沿之间被置位一个时钟周期且数据 data_in 放到 d 中。

一些其他任务可以按同样的方式定义：

- clr_counter_async：通过生成一个短的 reset 脉冲对计数器异步复位；
- clr_counter_sync：通过激活 syn_clr 信号一个时钟周期对计数器异步复位；
- count：使能计数器向上或向下计数一定数量的时钟周期；
- initialize：设置仿真初始值并生成一个复位脉冲。

利用这些程序，可以用更加抽象的方式生成测试向量：

```
initial
begin
   initialize( );
   count(12, 1);              // 向上计数12 个周期
   count(6, 0);               //向下计数6 个周期
   load_data(3'b011);         // 载入011
   count(2, 1);               // 向上计数2 个周期
   clr_counter_sync( );       // 同步清零
   cont(3, 1);                // 向上计数3 个周期
   clr_counter_sync( );       // 异步清零
   count(5, 1);               // 向上计数5 个周期
```

```
      $ stop;                          // 仿真停止
  end
```

完整的代码如示例 7.17 所示。

示例 7.17　测试向量生成器

```
module bin_gen
  #(parameter N = 8, T = 20)
  (
    output reg clk, reset,
    output reg syn_clr, load, en, up,
    output reg [N-1: 0] d
  );
  // 时钟
  // 时钟一直运行
  always
  begin
      clk = 1'b1;
      #(T/2);
      clk = 1'b0;
      #(T/2);
  end
  // 测试程序
  initial
  begin
      initialize();
      count(12, 1);                    // 向上计数12 个周期
      count(6, 0);                     // 向下计数6 个周期
      load_data(3'b011);               // 载入计数初值3
      count(2, 1);                     // 向上计数2 个周期
      clr_counter_sync();
      count(3, 1);                     // 向上计数3 个周期
      clr_counter_async();
      count(5, 1);                     // 向上计数5 个周期
       $ stop;                         //simulation 停止
  end
```

```
// ========================================
//task 定义
// ========================================
// 在时钟沿之间置reset 有效
task clr_counter_async();
begin
    @(negedge clk);                        // 等待下降沿
    reset = 1'b1;
    #(T/4);                                // 有效4/T 时间
    reset = 1'b0;
end
endtask
task initialize();
// 系统初始化
begin
    en = 0;
    up = 0;
    load = 0;
    syn_clr = 0;
    d = 3'b000;
    clr_counter_async();
end
endtask
// 置syn_clr 有效一个时钟周期
task clr_counter_sync();
begin
    @(negedge clk);                        // 等待下降沿
    syn_clr = 1'b1;    //clear 置为有效
    @(negedge clk);
    syn_clr = 1'b0;
end
endtask
// 载入寄存器
task load_data(input wire [N-1: 0] data_in);
begin
```

```
    @(negedge clk);                    // 等待下降沿
    load = 1'b1;
    d = data_in;
    @(negedge clk);
    load = 1'b0;
  end
  endtask
  // 向上(下)计数C 个周期
  task count(input integer C, input integer UP_DOWN);
  begin
    @(negedge clk);                    // 等待下降沿
    en = 1'b1;
    if (UP_DOWN == 1)                  // 如果up_down 为1，向上计数
      up = 1'b1;
    repeat(C) @(negedge clk);
    en = 1'b0;
    up = 1'b0;
  end
  endtask
endmodule
```

监视模块　监视模块监视并记录计数器的活动并验证相应操作。完整的代码如示例 7.18 所示。

示例 7.18　监视器

```
module bin_monitor
  #(parameter N = 3)
  (
   input wire clk, reset,
   input wire syn_clr, load, en, up,
   input wire [N-1:0] d,
   input wire max_tick, min_tick,
   input wire [N-1:0] q
  );
  reg [N-1:0] q_old, d_old, gold;
```

```
reg syn_clr_old, en_old, load_old, up_old;
reg [39:0] err_msg; //5 个字母的信息
initial  //开头
   $ display("time syn_clr/load/en/up   q\n");
always @ (posedge clk)
begin
    //_old: 上一个时钟沿采样的数据
    syn_clr_old  <= syn_clr;
    en_old  <= en;
    load_old  <= load;
    up_old  <= up;
    q_old  <= q;
    d_old  <= d;
    // 计算期望的"黄金"参考值
    if (syn_clr_old)
      gold = 0;
    else if (load_old)
      gold = d_old;
    else if (en_old & up_old)
      gold = q_old + 1;
    else if (en_old &  ~up_old)
      gold = q_old  -  1;
    else
      gold = q_old;
    // 错误信息
    if (q = = gold)
      err_msg = "     ";   // 结果通过
    else
      err_msg = "ERROR";    // 结果未通过
    //
    $ display("%5d,    %b%b%b%b   %d   %s",
              $ time, syn_clr, load, en, up, q, err_msg);
 end
endmodule
```

由于计数器是一个同步时序电路，因此，监视模块主要关注时钟上升时刻的活动。关键是检验计数器操作的正确性。由于被测试电路只是一个简单的记录器，可以从前一个采样沿记录采样的输入值和计数器状态，并判断新的计数器状态。例如，如果 syn_clr 的前一个采样值为 1，那么计数器会在下一个上升沿清零。

代码主要是一个 always 块，在时钟上升沿触发。包含三段。第一段使用非阻塞语句推荐寄存器，使用_old 后缀并存储从前一采样沿采样的数值。第二段用这些数值计数期望的计数器输出，即黄金参考。最后一段将期望的计数器输出值与实际输出值比较并显示采样的输入信号和计数器输出值。如果出现不匹配，则会产生一个 ERROR 信息。注意，在 Verilog 中，字符被当作 8bit 数字，这样 5 个字母的信息，err-msg，被声明为 reg[39：0]。

顶层模块 测试平台顶层模块的代码如示例 7.19 所示，设计遵循图 7-3 所示的框图。

示例 7.19 测试的顶层模块

```
timescale 1 ns/10 ps
module bin_counter_tb3();
  // 声明
localparam  T=20; // 时钟周期
  wire clk, reset;
  wire syn_clr, load, en, up;
  wire [2:0] d;
  wire max_tick, min_tick;
  wire [2:0] q;
  //uut 实例化
  univ_bin_counter #(.N(3))uut
    (.clk(clk), .reset(reset), .syn_clr(syn_clr),
      .load(load), .en(en), .up(up), .d(d),
      .max_tick(max_tick), .min_tick(min_tick), .q(q));
  // 测试向量生成器
  bin_gen #(.N(3), .T(20)) gen_unit
    (.clk(clk), .reset(reset), .syn_clr(syn_clr),
      .load(load), .en(en), .up(up), .d(d));
  //bin_monitor 实例化
  bin_monitor #(.N(3)) mon_unit
```

```
(.clk(clk), .reset(reset), .syn_clr(syn_clr),
  .load(load), .en(en), .up(up), .d(d),
  .max_tick(max_tick), .min_tick(min_tick), .q(q));
endmodule
```

除了波形，测试平台还会在控制台产生文本输出。

```
time syn_clr/load/en/up q
0       0000    x   ERROR
20      0011    0   ERROR
40      0011    0
60      0011    1
80      0011    2
100     0011    3
120     0011    4
140     0011    5
160     0011    6
180     0011    7
200     0011    0
220     0011    1
240     0011    2
260     0011    3
280     0000    4
300     0010    4
320     0010    3
340     0010    2
360     0010    1
380     0010    0
400     0010    7
420     0000    6
440     0011    6
460     0000    3
480     0011    3
500     0011    4
520     0000    5
540     1000    5
```

```
560    0000    0
580    0011    0
600    0011    1
620    0011    2
640    0000    3
660    0000    0    EEROR
680    0011    0
700    0011    1
720    0011    2
740    0011    3
760    0011    4
```

有 3 个 ERROR 信息。在系统初始化过程中，在 0 和 20 出现了 ERROR 信息，不是真正的错误。在 660 时刻出现的信息是由于 clr_counter_async( ) 操作所致，该操作在 640 和 660 时刻的采样沿之间产生一个短的异步脉冲。由于测试平台只监视同步活动，因此会错过异步复位并报告一个错误。

7.6 文献备注

由 Palnitkar 编著的《Verilog HDL，2nd edition》和 M. D Ciletti 编著的《Starter's Guide to Verilog2001》涵盖了 Verilog 语法和结构。IEEE Standard Verilog Hardware Description Language，IEEE Std 1364 - 2001 提供了关于有符号和无符号数据类型混合表达式调整的规则。J. Bergeron 编著的《Writing Testbenches：Functional Verification of HDL Models，2nd edition》一书对测试平台开发提供了详尽的讨论。C. E. Cummings 的论文《Nonblocking Assignments in Verilog Synthesis，Coding Styles That Kill!》为合理使用阻塞和非阻塞赋值提供了指南。

7.7 实验

7.7.1 使用阻塞和非阻塞赋值的移位寄存器

示例 7.20 所示代码是描述移位寄存器的三种尝试。通过三种尝试得到推断电路，并判断它们是否推断出了一个移位寄存器。

示例 7.20　实验 7.7.1 代码

```
module exp1
```

```
(
  input wire clk,
  input wire x0,y0,z0,
  output reg x3,y3,z3
);
reg x1,x2,y1,y2,z1,z2;
// 尝试1
always @ (posedge clk)
begin
     x1 <= x0;
     x2 <= x1;
     x3 <= x2;
end
// 尝试2
always @ (posedge clk)
begin
     y1 = y0;
     y2 = y1;
     y3 = y2;
end
// 尝试3
always @ (posedge clk)
begin
     z1 = z0;
     z3 = z2;
     z2 = z1;
end
endmodule
```

7.7.2 BCD 计数器的另一种代码风格

使用 7.2 节讨论的代码风格重新编写示例 4.18 中的 BCD 计数器，重新综合电路并验证其运行正确性。

7.7.3 FIFO 缓冲器的另一种代码风格

使用 7.2 节讨论的代码风格重新编写示例 4.20 中的 FIFO 缓冲器。重新综合电路并验证其运行正确性。

7.7.4 斐波纳契数电路的另一种代码风格

使用 7.2 节讨论的代码风格重新编写 6.3.1 节讨论的斐波纳契数电路。

7.7.5 双模式比较器

双模式比较器有两个 8bit 数据输入，a 和 b，可以是有符号或无符号整型数据。一个控制信号，mod，用于指示期望的模式。电路有一个输出 agtb，当翻译后的 a 值比 b 值大，则置为有效。

1）假设有符号数据类型允许使用，设计电路并形成代码；

2）综合电路并验证运行的正确性；

3）假设有符号数据类型在代码中不允许。重复步骤 1 和 2。

7.7.6 增加的二进制监视器

7.5.10 节的监视模块用于监视一个同步系统且只能检验时钟上升沿的活动。异步复位操作会被当作一个错误而报告。将异步操作考虑进来，修改监视电路。重新生成测试平台并执行仿真以验证其运行的正确性。

7.7.7 FIFO 缓冲器测试平台

按 7.5.10 节的例子设计一个紧凑的测试平台来验证 4.5.3 节讨论的 FIFO 缓冲器。测试向量生成器模块应生成不同组合的读和写操作，并使 FIFO 产生空和满条件。监视模块应持续监视数据写入并从缓冲器中回读，以检验操作的正确性。

第二部分

I/O 模块

第 8 章　UART

8.1　引言

通用异步收发传输器（UART），是一种用串行通信方式来传输并行数据的电路。UART 通常与 EIA（电子工业联盟）认定的 RS-232 标准同时使用，RS-232 标准详细说明了两个数据通信装置的电气、机械、功能及程序特性。RS-232 标准规定的电平与 FPGA 的 I/O（输入输出）电压不相同，因此需要在串口和 FPGA 的 I/O 口之间使用电平转换芯片。

S3 板有一个标准 9 针的 RS-232 端口。板子上包含必需的电压转换芯片并将 RS-232 的控制信号设置为与 PC 的串口匹配。标准的直通电缆用于连接 S3 板和 PC 的串口。S3 板使用 RS-232 标准，我们只需要关注 UART 电路的设计。

UART 包括一个发送器和一个接收器。发送器本质上是专用的移位寄存器，它并行的加载数据按照指定的速率将数据按位移出。另一方面，接收器按位接收并将数据重新组合。当串行线空闲的时候将串行线置 1。发送从起始位开始（起始位为 0），接着是数据位和可选的奇偶校验，最后是停止位（停止位为 1）。数据位可能是 6、7 或 8 位。奇偶校验位用于错误检测。若使用奇校验，当数据位有奇数个 1 时校验位设置为 0；若使用偶校验，当数据位有偶数个 1 时校验位设置为 0。停止位的个数可以是 1、1.5 或 2。图 8-1 为一个数据字的发送，这个数据字包括 8 位数据位、没有校验位和 1 位停止位。需要注意的是，发送时数据字的低位在前高位在后。

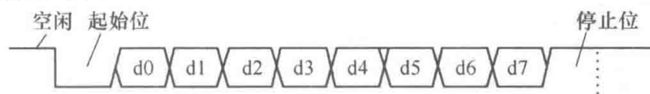


图 8-1　一个字节的传输

在串口线上没有时钟信息。在传输开始前，发送器和接收器须将包括波特率（即，每秒传输的位数）、数据位和停止位的数量和奇偶校验位的使用在内的参数提前设置好。通常使用的波特率为 2400bit/s、4800bit/s、9600bit/s 和 19 200bit/s。

在接下来的章节中我们将举例说明接收和发送子系统的设计。设计中设定

UART 接口的波特率为 19 200bit/s，有 8 位数据位和 1 位停止位，无校验位。

8.2 UART 接收子系统

由于输出信号没有时钟信息，接收模块只能按预先确定好的参数接收数据。我们使用过采样法估计发送数据位的中间点，接着在这些点处将数据位取出。

8.2.1 过采样步骤

最常用的采样速率为 16 倍的波特率，即每一个串行位被采样 16 次。如果在通信中使用 N 个数据位和 M 个停止位，过采样过程如下：

1）当传输信号为 0 时，即起始位的开始，开始采样计数；

2）当计数到 7 时，即计数到起始位的中间点时，计数器清零后重新计数；

3）当计数到 15 时，即到达第一个数据位的中间点时，取出输出值并将其锁存到移位寄存器中，同时计数器重新计数；

4）重复步骤 3）（$N-1$）次，接收余下的数据位；

5）如果用到奇偶校验位，重复步骤 3）一次获得校验位；

6）重复步骤 3）M 次获得停止位。

过采样法基本上履行了时钟信号的功能。它利用采样点来估计每一位的中间点，而不是像时钟信号那样用上升沿来判断输入信号的有效。当接收端没有精确地获得起始位的发起时间时，所估计的中间点的误差最多为波特率的 1/16。由于使用过采样，串行传输的波特率只能是系统时钟的一个很小的分频，因此，这个方法并不适用高数据传输速率。

UART 接受子系统的原理框图如图 8-2 所示，包括 3 个重要的部分：

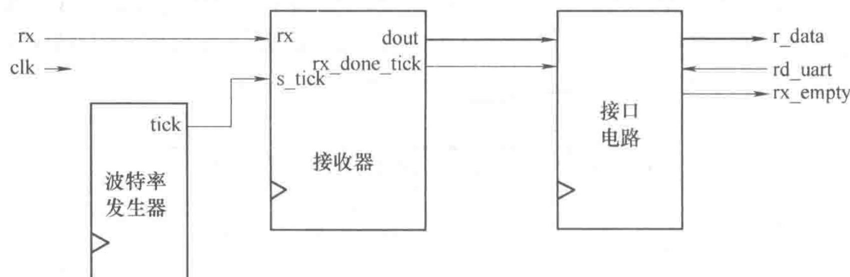


图 8-2　一个 UART 接收子系统的原理框图

- UART 接收器：通过过采样获得数据字的电路；
- 波特率发生器：产生采样点的电路；
- 接口电路：能提供缓存和在 UART 接收器和使用 UART 的系统之间状况

的电路。

8.2.2 波特率产生器

波特率产生器产生一个采样信号，它的频率是 UART 波特率的 16 倍。为了避免产生一个新的时钟域以及违反同步设计准则，对 UART 接收端来说，采样信号作为一个使能信号来使用，而不是作为一个时钟来使用，我们已在 4.3.2 节讨论这个问题。

对于 19 200bit/s 的波特率来说，采样速率应该是 307 200bit/s，由于时钟速率为 50MHz，因此需要一个模 163 计数器 $\left(即，\frac{50\times10^6}{307200}\right)$，即每 163 个时钟周期产生一个采样点。4.3.2 节中提到的计数器的 M 可以设置为 163。

8.2.3 UART 接收端

基于我们对过采样法的理解，可以得到 ASMD 图表，如图 8-3 所示。为了便于以后的修改，我们使用了两个常量。常量 D_BIT 表示数据位的个数，SB_TICK 表示停止位的个数，停止位个数为 1、1.5 和 2 时对应 SB_TICK 的值分别为 16、24、32。D_BIT 和 SB_TICK 在本设计中分别为 8 和 16。

图表中的步骤包含在 8.2.1 中，包含的 3 个状态：开始、数据和停止，分别代表着起始位、数据位和停止位。信号 s_tick 是由波特率发生器产生的使能信号，比特之间的间隔为 16 个计数点。只有 s_tick 为 1 时，FSMD 的状态才发生变化。寄存器 s 和 n 代表着两个计数器。寄存器 s 对采样点进行计数：在起始状态计数到 7，在数据状态计数到 15，在停止状态计数到 SB_TICK。寄存器 n 在数据状态下对接收到的数据位进行计数。对接收到的数据通过移位的方法组合到寄存器 b 中，状态信号 rx_done_tick 在接收进程结束后置一个时钟周期的 1，相关的代码如示例 8.1 所示。

示例 8.1 UART 接收器

```
...
module uart_rx
  #(
    parameter DBIT =8,          // #数据位
              SB_TICK =16   // #ticks 的停止位
  )
  (
    input wireclk, reset,
```

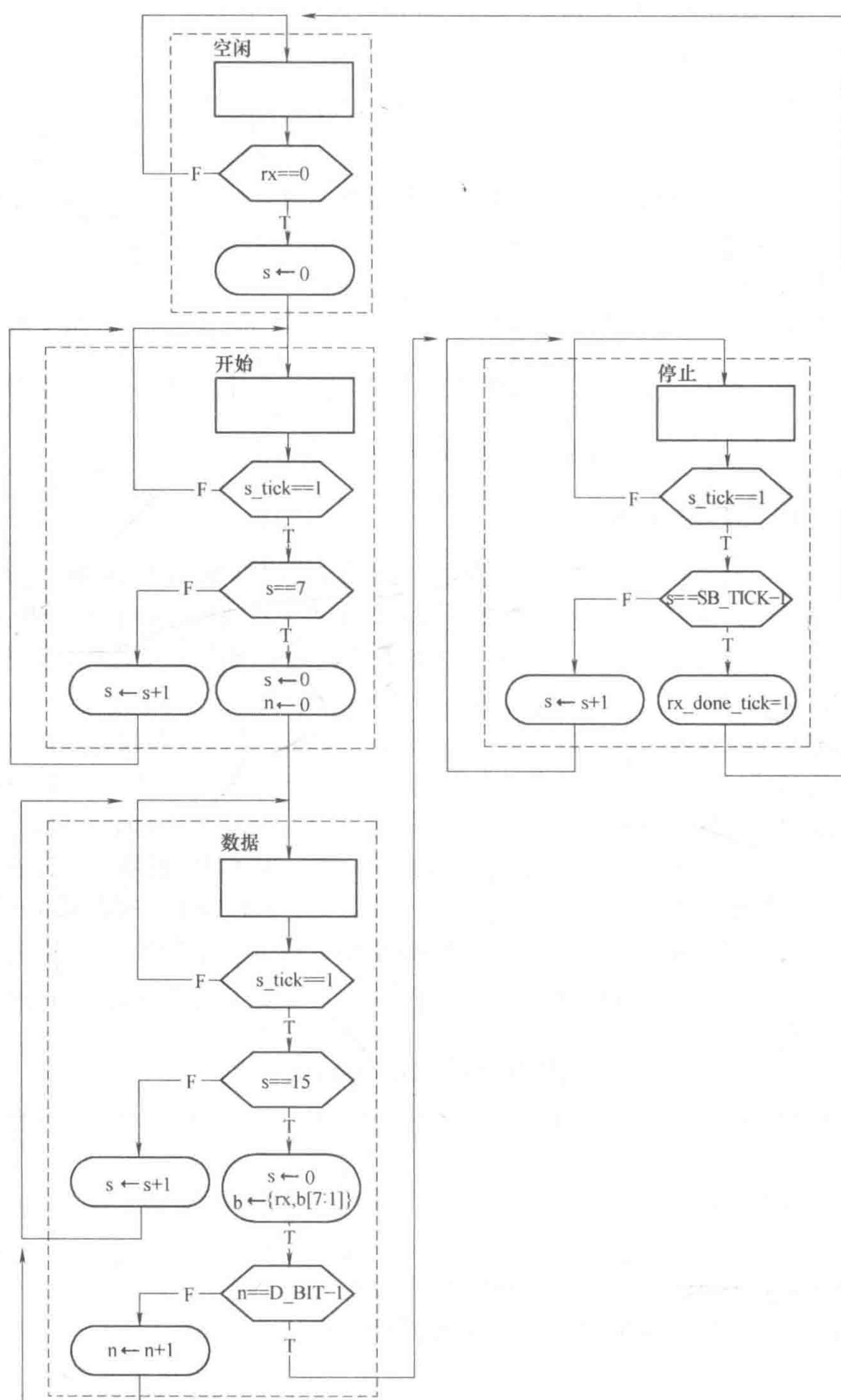


图 8-3　一个 UART 接收器的 ASMD 图

```
    input wire rx, s_tick,
    output reg rx_done_tick,
    output wire [7:0]dout
  );
  //符号处理数据的声明
localparam [1:0]
      idle =2'b00,
      start =2'b01,
      data =2'b10,
      stop =2'b11;
  //信号声明
  reg [1:0] state_reg, state_next;
  reg [3:0] s_reg, s_next;
reg [2:0] n_reg, n_next;
reg [7:0] b_reg, b_next;
  // 实体
  //FSMD 状态和数据寄存器
  always @ (posedge clk, posedge reset)
      if (reset)
        begin
            state_reg  <= idle;
            s_reg  <=0;
            n_reg  <=0;
            b_reg  <=0;
        end
      else
        begin
            state_reg  <= state_next;
            s_reg  <= s_next;
            n_reg  <= n_next;
            b_reg  <= b_next;
        end
  //FSMD 次态逻辑
  always @ *
  begin
```

```
state_next = state_reg;
rx_done_tick = 1'b0;
s_next = s_reg;
n_next = n_reg;
b_next = b_reg;
case (state_reg)
  idle:
      if (~rx)
        begin
            state_next = start;
            s_next = 0;
        end
  start:
      if (s_tick)
        if (s_reg == 7)
            begin
              state_next = data;
              s_next = 0;
              n_next = 0;
            end
        else
            s_next = s_reg + 1;
  data:
      if (s_tick)
        if (s_reg == 15)
            begin
              s_next = 0;
              b_next = {rx, b_reg[7:1]};
              if (n_reg == (DBIT - 1))
                  state_next = stop;
                else
                  n_next = n_reg + 1;
            end
        else
            s_next = s_reg + 1;
```

```
        stop:
            if (s_tick)
                if (s_reg = = (SB_TICK - 1))
                    begin
                        state_next = idle;
                        rx_done_tick = 1'b1;
                    end
                else
                    s_next = s_reg + 1;
    endcase
  end
  // 输出
  assign dout = b_reg;
endmodule
```

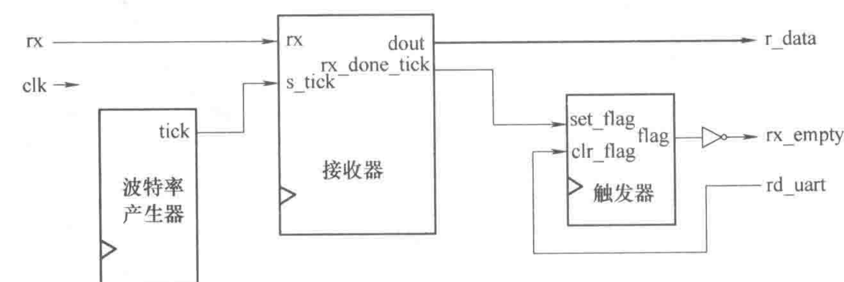
8.2.4 接口电路

在一个大的系统中，一个 UART 通常是一个用于串行数据传递的外围电路。主系统周期性地检索和处理接收到的数据。接收端接口电路有两个功能：第一，它提供一种机制来接收有效的数据并防止重复接收；第二，它为接收端和主系统之间提供一个缓存空间。通常有以下三种方案：

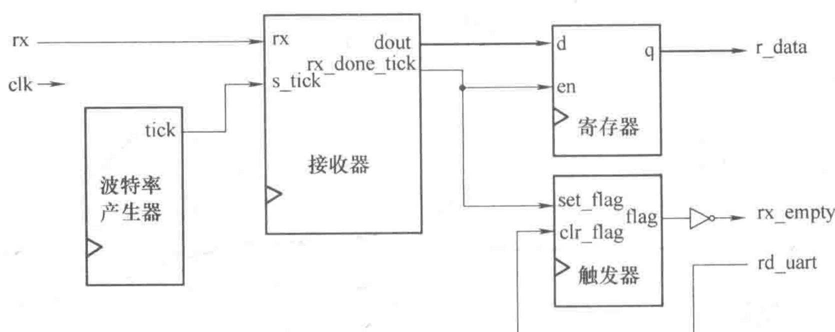
- 单指示触发器；
- 单指示触发器与单字缓冲器；
- 一个 FIFO 缓冲器。

注：rx_ready_tick 信号在一个数据字接收完后置一个时钟周期的高电平。

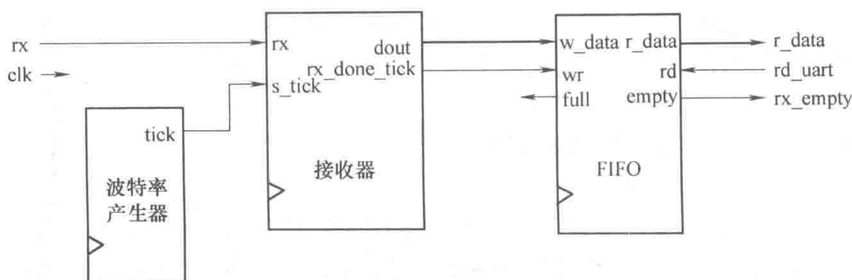
第一种方案用一个触发器来记录是否有新的数据字可用。触发器有两个输入信号，一个是置位端(set_flag)，可以使触发器输出 1；一个是清零端(clr_flag)，可以使触发器输出 0。rx_ready_tick 与置位端(set_flag)相连，当有一个新的数据字到来时将触发器置 1。主系统检测触发器的输出端来判断新的数据的有效性。clr_flag 信号在一个数据字接收完后置一个时钟周期的高电平。顶层模块图如图 8-4a 所示。为了保持一致性，在触发器的输出端增加一个反相器，输出信号为 rx_empty，用信号 rx_empty 高电平来表示输入信号是无效的。在这个系统中，主系统直接从 UART 的移位寄存器中接收数据，中间没有另外的缓存空间。如果在当前数据正在处理时(即，触发器一直处于高电平，即空状态)从外围接收到一个新的数据，正在处理的数据就会被覆盖掉。



a) 触发器



b) 触发器和一个单字缓存器



c) FIFO缓存器

图 8-4　UART 接收子系统的接口电路

为了增加缓冲，需要增加一个单字缓冲器，如图 8-4b 所示。当 rx_ready_tick 信号置为高电平时，缓存器将接收到的数据字存入缓冲区中，指示触发器也已经处于置位状态。接收器可以继续接收数据而不会覆盖前一个数据。这个系统的代码如示例 8. 2 所示。

示例 8. 2　指示触发器和缓冲器的接口

```
module flag_buf
```

```
#(parameter W =8) // #缓冲位
(
  input wireclk, reset,
  input wire clr_flag, set_flag,
  input wire [W-1: 0] din,
  output wire flag,
  output wire [W -1: 0]dout
);
//信号声明
reg [W-1: 0]buf_reg, buf_next;
reg flag_reg, flag_next;
// 实体
//FF 和寄存器
always @ (posedge clk, posedge reset)
    if (reset)
      begin
          buf_reg <=0;
          flag_reg <=1'b0;
      end
    else
      begin
          buf_reg <=buf_next;
          flag_reg <=flag_next;
      end
//次态逻辑
always @ *
begin
    buf_next =buf_reg;
    flag_next =flag_reg;
    if (set_flag)
      begin
          buf_next =din;
          flag_next =1'b1;
      end
    else if (clr_flag)
```

```
        flag_next = 1'b0;
  end
  // 输出逻辑
  assign dout = buf_reg;
  assign flag = flag_reg;
endmodule
```

第三种方案是 4.5.3 节中描述的 FIFO 缓存器。FIFO 缓存器可以提供更大的缓存空间，从而避免数据覆盖。我们可以根据主系统的数据要求来调整 FIFO 中数据位数。模块图如图 8-4c 所示。

信号 rx_ready_tick 与 FIFO 信号的写使能信号(wr)相连。当一个数据字接收完后，写信号(wr)置一个时钟周期的高电平并且相应的数据被写入 FIFO 中。主系统从 FIFO 读端口读出数据。当接收完一个字后，读信号(rd)置一个时钟周期的高电平并移出相应的数据字。FIFO 中的空信号(empty)表示是否有接收到的数据字。数据覆盖错误发生在 FIFO 满时，仍然接收数据。

8.3 UART 发送子系统

UART 发送子系统与接收子系统的机制是类似的。它包括一个 UART 发送器、波特率发生器和接口电路。接口电路与接收子系统类似，除了方向不一样：发送端的主系统置位触发器或写 FIFO，UART 发送器清除触发器或读 FIFO。

UART 发送器本质上来说是一个按照特定的速率把数据位一位一位输出的移位寄存器。速率可以通过波特率发生器来控制。因为不需要使用过采样法，发送端的波特率是接收端波特率的 1/16。不需要使用一个新的计数器，UART 发送端和接收端共用一个波特率发生器，使用内部的一个计数器来计数，每 16 个计数输出一个使能信号。

UART 发送的 ASMD 图表与 UART 接收类似。信号 tx_start 有效之后，ASMD 下载数据，逐步执行开始、数据和停止 3 个状态并输出数据。tx_done_tick 信号在发送完成后置一个时钟周期的高电平。缓存器 tx_reg 用来过滤毛刺。相关的代码如示例 8.3 所示。

示例 8.3　UART 发送器

```
module uart_tx
  #(
    parameter DBIT = 8,        // #数据位
```

```
            SB_TICK = 16  // #ticks 的停止位
  )
  (
    input wireclk, reset,
    input wire tx_start, s_tick,
    input wire [7: 0] din,
    output reg tx_done_tick,
    output wire tx
  );
  // 象征性的状态声明
localparam [1: 0]
      idle  =2'b00,
      start =2'b01,
      data  =2'b10,
      stop  =2'b11;
  // 信号声明
  reg [1: 0] state_reg, state_next;
  reg [3: 0] s_reg, s_next;
  reg [2: 0] n_reg, n_next;
  reg [7: 0] b_reg, b_next;
  reg tx_reg, tx_next;
  // 实体
  //FSMD 状态和数据寄存器
  always @ (posedge clk, posedge reset)
      if (reset)
        begin
            state_reg  <= idle;
            s_reg  <=0;
            n_reg  <=0;
            b_reg  <=0;
            tx_reg  <=1'b1;
        end
      else
        begin
            state_reg  <= state_next;
```

```
            s_reg  <= s_next;
            n_reg  <= n_next;
            b_reg  <= b_next;
            tx_reg  <= tx_next;
        end
//FSMD 次态逻辑和功能单元
always @ *
begin
    state_next = state_reg;
    tx_done_tick = 1'b0;
    s_next = s_reg;
    n_next = n_reg;
    b_next = b_reg;
    tx_next = tx_reg;
    case (state_reg)
      idle:
          begin
            tx_next = 1'b1;
            if (tx_start)
                begin
                  state_next = start;
                  s_next = 0;
                  b_next = din;
                end
          end
      start:
          begin
            tx_next = 1'b0;
            if (s_tick)
                if (s_reg = = 15)
                  begin
                      state_next = data;
                      s_next = 0;
                      n_next = 0;
                  end
```

```
                else
                    s_next = s_reg + 1;
            end
        data:
            begin
                tx_next = b_reg[0];
                if (s_tick)
                    if (s_reg = = 15)
                        begin
                            s_next = 0;
                            b_next = b_reg > > 1;
                            if (n_reg = = (DBIT - 1))
                                state_next = stop;
                            else
                                n_next = n_reg + 1;
                        end
                    else
                        s_next = s_reg + 1;
            end
        stop:
            begin
                tx_next = 1'b1;
                if (s_tick)
                    if (s_reg = = (SB_TICK - 1))
                        begin
                            state_next = idle;
                            tx_done_tick = 1'b1;
                        end
                    else
                        s_next = s_reg + 1;
            end
    endcase
end
// 输出
assign tx = tx_reg;
```

```
endmodule
```

8.4 UART 总系统简述

8.4.1 完整的 UART 核

通过结合接收和发送子系统，我们可以构造完整的 UART 核。顶层模块连接如图 8-5 所示。模块图表可以通过模块实例来描述，相应的代码如示例 8.4 所示。

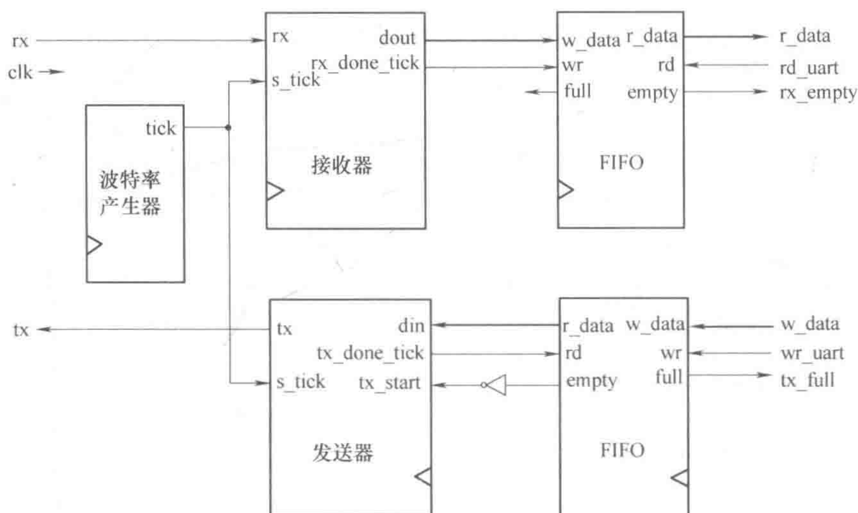


图 8-5　完整的 UART 框图

示例 8.4　UART 顶层描述

```
module uart
  #( // Default setting:
     //19 ,200 波特, 8 个数据位, 1 个停止位, 2^2 FIFO
     parameter DBIT = 8,       //#数据位
               SB_TICK = 16, //# 对停止位的时钟计时
                             // 对1/1.5/2 个停止位分别计时16/24/32
               DVSR = 163,    // 波特率因子
```

```
                                //DVSR =50M/(16 * 波特率)
                DVSR_BIT =8, //# DVSR 位数
                FIFO_W =2       //#FIFO 地址位数
                                //# FIFO 存储字数=2^FIFO_W
 )
 (
   input wireclk, reset,
   input wire rd_uart, wr_uart, rx,
   input wire [7:0] w_data,
   output wire tx_full, rx_empty, tx,
   output wire [7:0] r_data
 );
 //信号声明
 wire tick, rx_done_tick, tx_done_tick;
 wire tx_empty, tx_fifo_not_empty;
 wire [7:0] tx_fifo_out, rx_data_out;
 //实体
 mod_m_counter #(.M(DVSR), .N(DVSR_BIT)) baud_gen_unit
     (.clk(clk), .reset(reset), .q(), .max_tick(tick));
uart_rx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_rx_unit
     (.clk(clk), .reset(reset), .rx(rx), .s_tick(tick),
     .rx_done_tick(rx_done_tick), .dout(rx_data_out));
 fifo #(.B(DBIT), .W(FIFO_W)) fifo_rx_unit
     (.clk(clk), .reset(reset), .rd(rd_uart),
     .wr(rx_done_tick), .w_data(rx_data_out),
     .empty(rx_empty), .full(), .r_data(r_data));
 fifo #(.B(DBIT), .W(FIFO_W)) fifo_tx_unit
     (.clk(clk), .reset(reset), .rd(tx_done_tick),
     .wr(wr_uart), .w_data(w_data), .empty(tx_empty),
     .full(tx_full), .r_data(tx_fifo_out));
uart_tx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_tx_unit
     (.clk(clk), .reset(reset), .tx_start(tx_fifo_not_empty),
     .s_tick(tick), .din(tx_fifo_out),
     .tx_done_tick(tx_done_tick), .tx(tx));
 assign tx_fifo_not_empty = ~tx_empty;
```

```
endmodule
```

在 picoBlaze 源文件中(第 15 章中)，Xilinx 提供了具有相同功能的 UART 模块。和上述实现不同的是，该模块使用了更底层的 Xilinx 原语。可以理解为是用 Xilinx 专用组件的门级描述。因为设计者具备关于 Xilinx 器件的了解并且可以熟练地使用其中的结构，UART 的功能与本章中描述的相比更加高效。通过比较两者的语言复杂度及电路规模大小是有借鉴意义的。

8.4.2 UART 验证配置

1. 验证电路 我们使用一个回环电路和一个 PC 来验证 UART 的功能。模块图如图 8-6 所示。在该电路中，S3 板的串口与 PC 串口相连。当从 PC 中发出一帧数据，接收到的数据字存储在 UART 接收端中的 4 字 FIFO 缓存器中。在接收时(通过 r_data 端口)，数据字加 1 并返回给发送端(通过 w_data 端口)。当按下除颤交换机的按钮时，交换机会产生一个时钟周期的激励信号，它与 rd_uart 和 wr_uart 相连。当这个单时钟周期的激励信号产生时，从接收端的 FIFO 中读出一个数据帧，同时把这个数据帧写入到发送端的 FIFO 中。例如，我们可以在 PC 上输入字符 HAL，这样 3 个数据字就会被存储在接收端的 FIFO 中。接下来我们连续按 S3 板上的按钮 3 次，IBM 三个数据帧就会发送和显示出来。UART 的 r－data 端口也连接在 S3 板上的 8 个 LED 上，tx_full 和 rx_empty 两个信号将会被连在板子上的七段显示器上。相应的代码如示例 8. 5 所示。

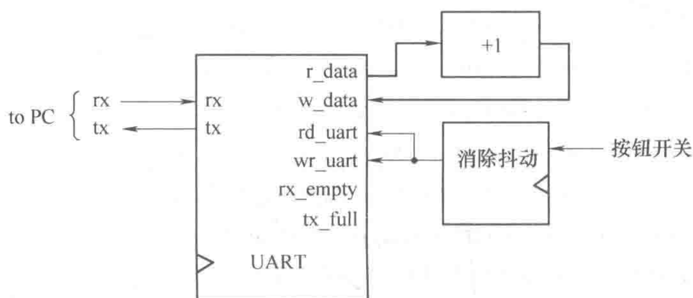


图 8-6　一个 UART 验证电路的框图

示例 8. 5　UART 验证电路

```
moduleuart_test
  (
    input wire clk, reset,
```

```
    input wire rx,
    input wire [2: 0]btn,
    output wire tx,
    output wire [3: 0] an,
    output wire [7: 0]sseg, led
  );
  // 信号声明
  wire tx_full, rx_empty, btn_tick;
wire [7: 0] rec_data, rec_data1;
// 实体
  //uart 实例
uart uart_unit
      (.clk(clk), .reset(reset), .rd_uart(btn_tick),
      .wr_uart(btn_tick), .rx(rx), .w_data(rec_data1),
      .tx_full(tx_full), .rx_empty(rx_empty),
      .r_data(rec_data), .tx(tx));
// 去抖电路实例
debounce btn_db_unit
      (.clk(clk), .reset(reset), .sw(btn[0]),
      .db_level(), .db_tick(btn_tick));
  // 递增数据环回
  assign rec_data1 = rec_data + 1;
  //LED 显示
  assign led = rec_data;
  assign an = 4'b1110;
  assignsseg = {1'b1, ~tx_full, 2'b11, ~rx_empty, 3'b111};
endmodule
```

2. Windows 超级终端 Windows 超级终端程序可以作为一个虚拟终端与 S3 板互连，为了与定制 UART 兼容，需要配置波特率为 19 200bit/s，8 个数据位，一个停止位并且无校验位。基本步骤如下：

1)选择“开始”→“所有程序”→“附件”→“通信”→“超级终端”。显示一个超级终端对话框。

2)把连接命名为 fpga_192。单击“确定”。这个连接就被保存下来并且可以被调用。

3）显示一个"连接到"对话框。点击"连接时使用"一栏并选择需要的串口（例如，COM1）。单击"确定"。

4）显示一个端口设置对话框。配置如下：

- 波特率：19 200bit/s；
- 数据位：8；
- 校验位：无；
- 停止位：1；
- 接下来的控制：无；

单击"确定"。

5）选择"文件"→"属性"→"设置"。单击"ASCII 码设置"并且将"本地回显键入的字符"勾选。双击"确定"。这样就会在屏幕上显示输入的值。

超级终端程序已经设置完成并可以和 S3 板通信。我们可以键入少量的字符然后观察 S3 板上的 LED。注意：存入 FIFO 中的所有的数据只有第一个字可以被显示。按"pushbutton"这个键后，第一个数据帧就会被读出，这个字符加 1 后的字符就会循环到 PC 的串口端并且在超级终端窗口显示。FIFO 的 full 和 empty 两个状态可以通过连续接收和发送超过 4 个数据字来验证。

3. ASCII 码　在超级终端 1 中，数据字是通过 ASCII 码的方式发送的，它包含 7bit 并且共有 128 个码：常见字母、数字、标点符号及控制字符。字符及编码（十六进制格式）见表 8-1。非打印字符如括号中所述，例如（del）。一些非打印字符在接收到后会产生如下效果：

- （nul）：空字节，全为 0；
- （bel）：产生铃响（如果条件允许）；
- （bs）：回格键 backspace；
- （ht）：Tab 键；
- （nl）：换行键；
- （vt）：垂直键；
- （np）：换页；
- （cr）：carriage return；
- （esc）：escape；
- （sp）：space；
- （del）：删除键，也可以表示全 1 状态。

因为我们在很多工程和试验中用 PC 的串口和 S3 板通信，以下几点有助于我们操作和处理 ASCII 码：

- 当第一个十六进制数为 0x0 或 0x1 时，相应的 ASCII 码为控制字符；
- 当第一个十六进制数为 0x2 或 0x3 时，相应的 ASCII 码为一个数字或标点

符号；

- 当第一个十六进制数为0x4 或0x5 时，相应的ASCII 码通常是一个大写字母；
- 当第一个十六进制数为0x6 或0x7 时，相应的ASCII 码通常是一个小写字母；
- 如果第一个十六进制数为0x3 时，低位的十六进制数相应的ASCII 码通常为十进制数；
- 单个大小写字母是有区别的，可以通过加(或减)'0x20'来相互转化。

注意ASCII 码只有7 位，但是一个数据字节通常有8 位(即，一个字节)。PC 使用一种扩展设置，当最高位为1 时，这些字符表示的是一些特殊的图形标志。这种编码并不是ASCII 码标准的一部分。

8.5 定制一个UART

上述章节中介绍的UART 都是根据特定的配置来定制的。该设计和代码可以通过很简单的修改来满足其他的需求：

- 波特率，波特率通过波特率发生器的采样频率来控制，频率可以通过改变模m 计数器的参数M 来改变；
- 数据位的个数，数据位的个数可以通过改变寄存器n_reg 的上限值来改变，即改变代码中常量DBIT 的值；
- 奇偶校验位，校验位可以通过在数据和停止状态之间加如一个校验状态来实现，这个状态可以添加在图8-3 中；
- 停止位的个数，数据位的个数可以通过改变寄存器s_reg 的上限(常量SB_TICK)值来实现，SB_TICK 可以为16、24、32，分别对应停止位的个数为1、1.5、2；
- 错误检查，3 种类型的错误可以在UART 接收子系统中发现：

——奇偶错误，如果数据帧中包含校验位，接收端可以检查校验位的正确性。

——帧错误，接收端可以检查停止位的值。如果不是1，那么这一帧就是错误的；

——缓存器数据覆盖错误，这种情况发生在主系统没有及时移出接收端的数据时。UART 接收端可以检查缓存器中flag_reg 信号或FIFO 的满标志位(即，当rx_done_tick 信号为1 时)。数据覆盖发生在flag_reg 信号被置位或FIFO 的满标志位为1 时。

8.6 文献备注

尽管 RS-232 标准已经很经典了，它仍然广泛应用于在两个器件之间提供简单、可靠、低速率的通信。Wikipedia 网站上有一篇很好的概述文章和一些有用的链接(搜索时输入关键字 RS232)。Jan Axelson 所完成的串口可以为连接硬件器件和 PC 提供一些参考。

8.7 实验

8.7.1 具备所有特征的 UART

具有可选择性的定制 UART 应当包含设计中的所有特征而且可以动态地根据需要来对 UART 进行配置。一个包含所有特性的 UART 需要使用额外的输入来配置波特率、校验位的类型、数据位的个数及停止位的个数。UART 同样包括错误信号。除了示例 8.4 中 uart_top 的 I/O 信号，还需要增加以下 I/O 端口信号：

- bd_rate：输入信号，2bit，用来明确波特率，可以是 1200 bit/s、2400 bit/s、4800 bit/s 或 9600bit/s；
- d_num：输入信号，1bit，用来明确数据位的个数，可以是 7 或 8；
- s_num：输入信号，1bit，用来明确停止位的个数，可以是 1 或 2；
- par：输入信号，2bit，用来明确需要的校验位类型，可以是无校验位、奇校验位或偶校验位；
- err：输出信号，3bit，用来显示校验位错误、帧错误或数据覆盖。

根据以下步骤来生成该电路：

1) 修改图 8-3 中的 ASMD 图来满足上述需求；
2) 根据 ASMD 图修改 UART 接收端的代码；
3) 根据需求修改 UART 发送端的代码；
4) 修改顶层 UART 代码和验证电路，用板子上的开关实现增加的输入，并用 3 个 LED 灯来代表错误输出信号，综合验证电路；
5) 在超级终端程序上创建不同的配置并验证 UART 的功能。

8.7.2 拥有波特率自动检测功能的 UART

最常用的串行连接数据位的个数是 8 位，即一个字节。当我们用 ASCII 码进行通信时(即，在超级终端窗口测试时)，只用了低 7 位，而高位为 0。如果一个 UART 的配置为 8 位数据位、1 位停止位及无校验位，接收到的数据形式如下：

0_dddd_ddd0_1,d代表数据位可以是0或1。假如数据帧之间有充足的时间，我们可以通过测量第一个0和最后一个0之间的间隔来确定波特率。基于以上方法，我们可以产生一个拥有波特率自动检测功能的UART。这样的话，发送端可以先发送一个ASCII码用于波特率检测再继续执行正常的功能，接收端系统用第一个数据字来决定波特率产生器的波特率，从而进行后续数据帧的传输。

假设一个UART的配置为8位数据位、1位停止位及无校验位，并且波特率可以是4800 bit/s、9600bit/s或19 200bit/s。UART的接收端应该有两种工作模式。开始在检测模式，等待第一个字。接收到第一个字后波特率也就确定了，接着接收端进入正常模式，即UART的功能也进入一个正常模式。根据以下步骤来生成：

1)绘出波特率自动检测电路的ASMD图；

2)根据ASMD图产生VHDL代码，使用S3板上的3个LED灯来显示接收到信号的波特率；

3)修改UART，使其包含3种波特率：4800 bit/s、9600bit/s及19200bit/s。可以使用一个寄存器来代表波特率的约数并且根据需求使用相应的波特率；

4)创建一个顶层的FSMD表来记录状态，从而控制并整合波特率检测电路及UART接收端的功能，使用S3板上的转换按钮来强制UART进入检测模式；

5)修改顶层UART代码和验证电路，综合验证电路；

6)在超级终端程序上创建不同的配置并验证UART的功能。

8.7.3 拥有波特率校验位自动检测功能的UART

除波特率外，我们假设校验位也是可以自动检查的，可以是无校验位、奇校验位或偶校验位。扩展上面的波特率检测电路来实现校验位配置检测。重复试验8.7.2。

8.7.4 UART控制的秒表

参考试验4.7.6中增强性的秒表，秒表的功能由S3板上的3个开关键控制，通过UART串口，我们可以用PC的超级终端程序来发送命令及接收秒表的时间：

- 当接收到一个c或C(clear)的ASCII码时，秒表终止当前计数，清零，设置计数方向为“up”；
- 当接收到一个g或G(go)的ASCII码时，秒表开始计数；
- 当接收到一个p或P(pause)的ASCII码时，计数暂停；
- 当接收到一个u或U(up - down)的ASCII码时，计数方向翻转；
- 当接收到一个r或R(receive)的ASCII码时，秒表向PC发送当前时间，

时间应显示为“DD. D”，D 代表十进制数；

● 当接收到其他的 ASCII 码时，不处理。

设计一个新的秒表，综合电路，连接 PC，用超级终端程序来验证功能。

8.7.5　UART 控制的 LED 标语

参考试验 4.7.5 中 LED 移动标语。通过 UART 串口，我们可以用 PC 的超级终端程序来控制 LED 灯的功能，并修改标语的数字：

● 当接收到一个 g 或 G(go)的 ASCII 码时，LED 开始移动；

● 当接收到一个 p 或 P(pause)的 ASCII 码时，移动暂停；

● 当接收到一个 d 或 D(direction)的 ASCII 码时，移动方向翻转；

● 当接收到十进制数字的 ASCII 码时，标语将会被修改，标语可以看作是一个十字的 FIFO 缓冲器。新的数字将插入在标语的开始端(即最左端)，最右端的数字将会被移出丢弃。

● 当接收到其他的 ASCII 码时，不处理；

设计一个新的 LED 灯移动标语，综合电路，连接 PC，用超级终端程序来验证功能。

第 9 章　PS2 键盘

9.1　引言

PS2 端口在 IBM 个人计算机的个人系列 /2 中被引入。在键盘、鼠标与主机的通信中，它是一个被广泛支持的接口。PS2 端口使用两条线进行通信。一条是串行传输的数据线，另一条传输时钟信息，此信息详细指出何时数据有效及何时可以接收数据。此时钟信息以包含一个起始位，8 个数据位，一个奇偶校验位和一个停止位的 11 - 位‘包’的方式进行传输。虽然一个键盘和鼠标的包的基本格式是相同的，但是它们的数据位的解释却不一样。FPGA 的原型板有一个 PS2 端口并作为主机被使用，在这章中我们将讨论键盘接口并在第 10 章中讨论鼠标接口。

PS2 端口的通信是双向的，主机可以向键盘和鼠标发送指令，用来设置特定参数。对我们的目的而言，PS2 键盘的双向通信是不需要的，因此我们只讨论从键盘到原型板的单方向通信。双向的设计在第 10 章的鼠标接口中考察。PS2 端口的时序图如图 9-1 所示。

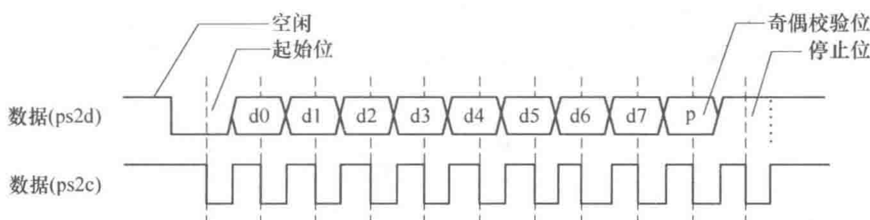


图 9-1　PS2 端口的时序图

9.2　PS2 接收子系统

9.2.1　PS2 端口的物理层接口

除了数据线和时钟，PS2 端口还包括与电源(即，Vcc)和地相连的线。电源由主机提供。最初的 PS2 端口，Vcc 是 5V，数据输出线和时钟是开放的收集器。但是，现在大部分的键盘和鼠标都可以在 3.3V 的电压下正常工作。对于一个老

版的键盘和鼠标，可以通过转换 S3 板上的 J2 跳线得到 5V 的电压。由于 FPGA 的 I/O 引脚可以承受 5V 的输入，所以 FPGA 仍能正常地运行。

9.2.2 设备到主机的通信协议

PS2 设备和主机通过包进行通信。从 PS2 设备到主机的传输时序图如图 9-1 所示，图中数据和时钟信号分别标为 ps2d 和 ps2c。

数据以串行模式进行传输，格式与 UART 相似。传输以一个起始位开始，接着是 8 个数据位和一个奇偶校验位，最后是一个停止位。与 UART 不同，PS2 的时钟信息是在一个单独的时钟信号(ps2c)上传输。ps2c 信号的下降沿表示 ps2d 上的相应位有效，并可以得到此位数据。ps2c 信号的时钟周期应在 60 ~ 100μs 之间(10 ~ 16.7kHz)，并且 ps2d 信号在 ps2c 信号的下降沿前后至少 5μs 内应保持稳定。

9.2.3 设计和代码

PS2 端口的接收子系统的设计有点类似 UART 接收器的设计。ps2c 信号的下降沿作为返回数据的参考点，代替了过采样策略。子系统包含一个检测下降沿的电路，它在 ps2c 信号的下降沿产生一个单时钟周期的脉冲，同时接收器移入并重新装配串行数据。

在第 5.3.1 节中讨论的边沿检测电路可以被用来检测下降沿并生成一个使能脉冲。然而，由于存在潜在噪声和跳变延迟的情况，所以需要增加一个简单的滤波电路用于除去干扰脉冲。这个滤波器的代码如下：

```
always @(posedge clk, posedge reset)
 . . .
        filter_reg <= filter_next;
 . . .
  //1-bit 移位器
assign filter_next = {ps2c, filter_reg[7:1]};
  //"过滤器"
assign f_ps2c_next = (filter_reg == 8'b11111111) ? 1'b1;
                     (filter_reg == 8'b00000000) ? 1'b0;
                     f_ps2c_reg;
```

这个电路由一个 8 位的移位寄存器构成，在接收到连续的 8 个 1 或者 0 的时候返回一个 1 或 0。任何一个比 8 个时钟周期短的干扰脉冲都将被忽略掉。接着，滤波器的输出信号会反馈给下降沿检测电路。

接收器的 ASMD 图如图 9-2 所示。接收器在 Idle 状态时被初始化。接收器包

含一个额外的控制信号，rx_en，它常被用来使能(或者不使能)接收操作。这个信号的目的是协调双向操作。对于键盘接口，它被置为1。

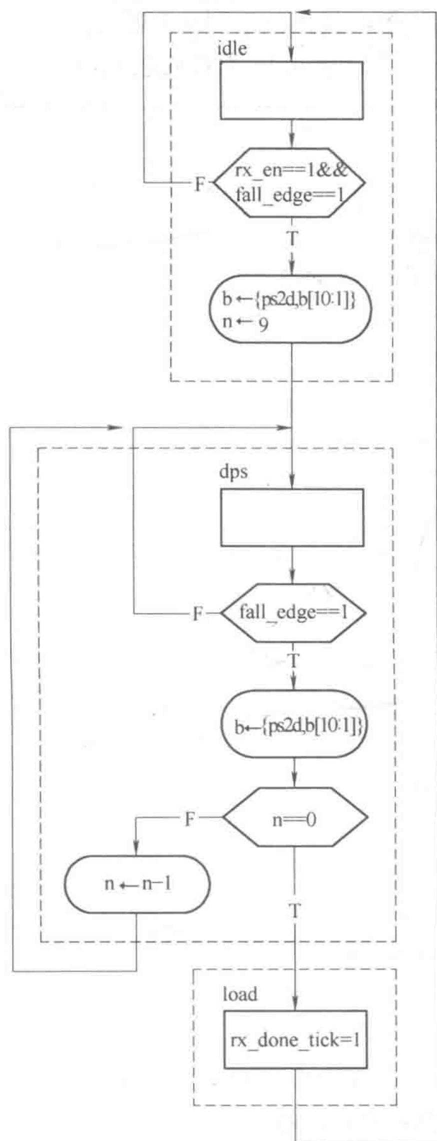


图9-2 PS2接收端口的ASMD图

在第一个下降沿脉冲到来后，rx_en信号被置为有效，FSMD移入开始位并转换到dps状态。由于接收的数据有固定格式，因此可以在一个单独的状态中移10位而不用将数据、奇偶和停止状态分开。接着，FSMD转换到加载状态，在这

个状态中，提供一个额外的时钟周期用于完成停止位的移位，在这个周期中 psrx_done_tick 信号被置为有效。HDL 代码包含滤波器电路和一个 FSMD 两部分，它遵循图 9-2 的 ASMD 图。HDL 代码如示例 9.1 所示。

示例 9.1　PS2 接收端口

```
module ps2_rx
  (
    input wire clk, reset,
    input wire ps2d, ps2c, rx_en,
    output reg rx_done_tick,
    output wire [7: 0] dout
  );
  // 符号状态声明
  localparam [1: 0]
      idle =2'b00,
      dps =2'b01,
      load =2'b10;
  // 信号声明
  reg [1: 0] state_reg, state_next;
  reg [7: 0] filter_reg;
  wire [7: 0] filter_next;
  reg f_ps2c_reg;
  wire f_ps2c_next;
  reg [3: 0] n_reg, n_next;
  reg [10: 0] b_reg, b_next;
  wire fall_edge;
  // 实体
  // ==================================================
  // 生成PS2c 下降沿脉冲和滤波器
  // ==================================================
  always @ (posedge clk, posedge reset)
        filter_reg  <= filter_next;
  //1bit 移位器
  assign filter_next = {ps2c, filter_reg[7: 1]};
  //"过滤器"
```

```
assign f_ps2c_next = (filter_reg = =8'b11111111) ? 1'b1 :
                    (filter_reg = =8'b00000000) ? 1'b0 :
                    f_ps2c_reg;
assign fall_edge = f_ps2c_reg & ~f_ps2c_next;
// ================================================================
//FSMD
// ================================================================
//FSMD 状态和数据寄存器
always @ (posedge clk, posedge reset)
    if (reset)
      begin
          state_reg  <= idle;
          n_reg  <=0;
          b_reg  <=0;
      end
    else
      begin
          state_reg  <= state_next;
          n_reg  <= n_next;
          b_reg  <= b_next;
      end
//FSMD 次态逻辑
always @ *
begin
    state_next = state_reg;
    rx_done_tick = 1'b0;
    n_next = n_reg;
    b_next = b_reg;
    case (state_reg)
      idle:
          if (fall_edge & rx_en)
            begin
                // 在转移起始位
                b_next = {ps2d, b_reg[10: 1]};
                n_next = 4'b1001;
```

```
                state_next = dps;
            end
      dps: //8 数据位+1 奇偶校验位+1 停止位
         if (fall_edge)
            begin
                b_next = {ps2d, b_reg[10: 1]};
                if (n_reg ==0)
                   state_next = load;
                else
                   n_next = n_reg-1;
            end
      load: //1 个额外的时钟未完成最后一次移位
         begin
            state_next = idle;
            rx_done_tick = 1'b1;
         end
    endcase
  end
  // 输出
  assign dout = b_reg[8: 1]; // 数据位
endmodule
```

以上描述了没有错误监测的电路。更鲁棒的设计应检查开始位、奇偶位、停止位，并且应当包含有监视时钟从而防止键盘锁死在一个错误的状态。这在本章最后被留作实验。

9.3 PS2 键盘的扫描码

9.3.1 扫描码概述

键盘由一个按键矩阵和一个嵌入式微型控制器组成，这个微型控制器监听（即，扫描）按键的动作，并因此发出扫描码。以下三种类型的按键动作会被监测到：

- 当按键按下的时候，输出按键的建立码；
- 当按键持续保持按下状态时，此情况被识别为连续键入，以一定的频率

重复输出建立码，默认情况下，PS2键盘在按键保持按下0.5s后每100ms输出建立码；

● 当按键被释放时，输出按键的停止码。

PS2键盘主要部分的建立码如图9-3所示。它通常为1字节的宽度，并且用两个十六进制的数表示。例如，A按键的建立码为1C。在传输的时候，这个码将通过一个包进行传递。少数具有特殊意义的按键(即扩展按键)的建立码，可以有2~4字节。这些按键的一部分如图9-3所示。例如，右侧的向上键的建立码是E075。需要多个包来传输。一般按键的停止码由F0后跟按键的建立码组成。例如，按键A的停止码为F01C。

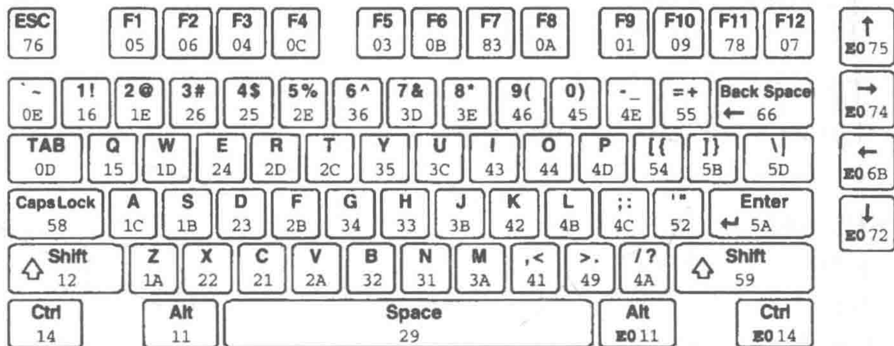


图9-3 PS2键盘的扫描编码

PS2键盘根据按键的行为输出一个码的序列。例如，当我们按下并释放按键A时，键盘先输出建立码，接着输出停止码。

1C F0 1C

如果我们按下按键并在保持一段时间后释放，建立码将被输出多次：

1C 1C 1C … 1C F0 1C

多个按键可以同时被按下。例如，我们先按下"Shift"键(它的建立码是12)，接着，按下"A"键，然后，释放"A"键，释放"Shift"键。输出码的序列遵循两个按键的建立码和停止码：

12 1C F0 1C F0 12

上述序列是我们怎样正常的获得大写字母A。需要注意的是，没有用来区别小写或大写键的码。由主设备负责跟踪"Shift"键是否已经按下，并根据此情况决定是哪个事件。

9.3.2 扫描码监听电路

扫描码监听电路监听接收包的到达并在PC的超级终端窗口显示扫描码。基

本的设计方法是首先将接收的扫描码分解为两个 4 位，并将它们看作是两个十六进制的数，接着，将两个数转换为 ASCII 码值，并将 ASCII 码通过 UART 发送给 PC。接收到的扫描码将像先前例子中的序列一样被显示。程序如示例 9.2 所示。

示例 9.2　PS2 键盘扫描编码的监听电路

```
module kb_monitor
  (
    input wire clk, reset,
    input wire ps2d, ps2c,
    output wire tx
  );
  // 常量声明
  localparam SP = 8'h20; //ASCII 空间
  // 符号状态声明
   localparam [1: 0]
      idle   =2'b00,
      send1 =2'b01,
      send0 =2'b10,
      sendb =2'b11;
  // 信号声明
  reg [1: 0] state_reg, state_next;
  reg [7: 0] w_data, ascii_code;
  wire [7: 0] scan_data;
  reg wr_uart;
  wire scan_done_tick;
  wire [3: 0] hex_in;
  // 实体
  // =========================================
  // 实例
  // =========================================
  //PS2 接收器实例
  ps2_rx ps2_rx_unit
      (.clk(clk), .reset(reset), .rx_en(1'b1),
      .ps2d(ps2d), .ps2c(ps2c),
      .rx_done_tick(scan_done_tick), .dout(scan_data));
```

```
//UART 实例
uart uart_unit
    (.clk(clk), .reset(reset), .rd_uart(1'b0),
    .wr_uart(wr_uart), .rx(1'b1), .w_data(w_data),
    .tx_full(), .rx_empty(), .r_data(), .tx(tx));
//===================================================
//FSM 发送3 个ASCII 字符
//===================================================
//状态寄存器
always @ (posedge clk, posedge reset)
  if (reset)
      state_reg <= idle;
  else
      state_reg <= state_next;
//次态逻辑
always @ *
begin
    wr_uart = 1'b0;
    w_data = SP;
    state_next = state_reg;
    case (state_reg)
      idle:
        if (scan_done_tick) //收到的扫描码
            state_next = send1;
      send1: //发送高十六进制数字
        begin
            w_data = ascii_code;
            wr_uart = 1'b1;
            state_next = send0;
        end
      send0: //发送低十六进制数字
        begin
            w_data = ascii_code;
            wr_uart = 1'b1;
            state_next = sendb;
```

```
        end
      sendb:    // 发送空白字符
        begin
          w_data = SP;
          wr_uart = 1'b1;
          state_next = idle;
        end
    endcase
end
// =========================================================
// 扫描码到ASCII 显示
// =========================================================
// 均分扫描码为两个4 位十六进制
assign hex_in = (state_reg == send1)? scan_data[7:4] :
                                      scan_data[3:0];
// 十六进制数字到ASCII 码
always @ *
case (hex_in)
    4'h0: ascii_code = 8'h30;
    4'h1: ascii_code = 8'h31;
    4'h2: ascii_code = 8'h32;
    4'h3: ascii_code = 8'h33;
    4'h4: ascii_code = 8'h34;
    4'h5: ascii_code = 8'h35;
    4'h6: ascii_code = 8'h36;
    4'h7: ascii_code = 8'h37;
    4'h8: ascii_code = 8'h38;
    4'h9: ascii_code = 8'h39;
    4'ha: ascii_code = 8'h41;
    4'hb: ascii_code = 8'h42;
    4'hc: ascii_code = 8'h43;
    4'hd: ascii_code = 8'h44;
    4'he: ascii_code = 8'h45;
    default: ascii_code = 8'h46;
  endcase
```

```
endmodule
```

使用FSM来控制全部的操作。当接收到一个新的扫描码(通过将scan_done_tick信号置为有效表示)时，启动UART操作。FSM在sendl、send0和sendb状态之间循环，在这些状态中将大写十六进制数，小写十六进制数和空格的ASCII码写入UART。上章介绍UART有一个四字的FIFO，因此不会产生溢出。注意，不使用UART的接收器，响应端口被映射到常量或者被留空。

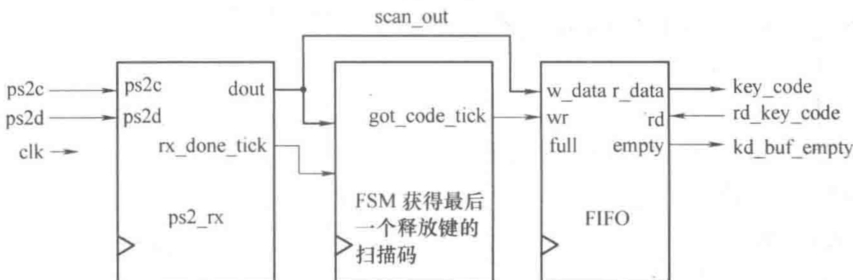


图9-4 键盘最后一个释放键电路的结构图

9.4 PS2键盘接口电路

正如9.3.1节所讨论的，即使是简单的键盘行为，也会发送一个包的序列。如果我们想要覆盖所有可能的组合，将会变得非常复杂。在这一节中，我们假设一次只按下和释放一个常规键，并设计一个电路返回这个键的建立码。这个设计提供了向原型板发送一个字母或数字的简单方法。

9.4.1 基本设计与HDL代码

键盘电路与UART相同是一个系统的外围电路，它需要一个与主系统通信的装置。已在第8.2.4节中讨论的标志和缓存结构同样可以用在键盘电路中。在这个设计中，我们使用一个四字的FIFO作为接口。

顶层的示意图如图9-4所示。它由PS2接收器、FIFO缓存和FSM控制器组成。基本想法是使用FSM追踪F0包的停止码。在它被接收后，下一个包应是这个键的建立码，这个包将被写入FIFO缓存器。注意，这个结构不能用于扩展键，因为它们的建立码包含多个包。相应的HDL代码如示例9.3所示。

示例 9.3　PS2 键盘最后一个释放键的电路

```
module kb_code
  #(parameter W_SIZE = 2)   //FIFO 中存储2^W_SIZE 个字
  (
    input wire clk, reset,
    input wire ps2d, ps2c, rd_key_code,
    output wire [7:0] key_code,
    output wire kb_buf_empty
  );
  // 常量声明
  localparam BRK = 8'hf0; // 断码
  // 符号状态声明
  localparam
      wait_brk = 1'b0,
      get_code = 1'b1;
  // 信号声明
  reg state_reg, state_next;
  wire [7:0] scan_out;
  reg got_code_tick;
  wire scan_done_tick;
  // 实体
  // =====================================================
  // 实例
  // =====================================================
  //PS2 接收器实例
  ps2_rx ps2_rx_unit
      (.clk(clk), .reset(reset), .rx_en(1'b1),
      .ps2d(ps2d), .ps2c(ps2c),
      .rx_done_tick(scan_done_tick), .dout(scan_out));
  //fifo 缓存器实例
  fifo #(.B(8), .W(W_SIZE)) fifo_key_unit
    (.clk(clk), .reset(reset), .rd(rd_key_code),
      .wr(got_code_tick), .w_data(scan_out),
      .empty(kb_buf_empty), .full(),
```

```
    .r_data(key_code));
// ==========================================================
// 收到F0 后得到扫描码的有限状态机
// ==========================================================
// 状态寄存器
always @(posedge clk, posedge reset)
  if (reset)
        state_reg <= wait_brk;
     else
        state_reg <= state_next;
// 次态逻辑
always @ *
begin
     got_code_tick = 1'b0;
     state_next = state_reg;
     case (state_reg)
        wait_brk:    // 等待F0 的断码
     if (scan_done_tick = = 1'b1 && scan_out = = BRK)
           state_next = get_code;
        get_code:    // 得到以下扫描码
     if (scan_done_tick)
        begin
           got_code_tick = 1'b1;
           state_next = wait_brk;
        end
     endcase
end
endmodule
```

代码的主要部分是 FSM，它用来扫描停止码，并协调两个模块的操作。它在 wait_brk 状态连续地检测接收到的包。当检查到 F0 包后，转到 get_code 状态并等待下一个包（它是这个键的建立码）。接着，在一个时钟周期内 FSM 将 code_done_tick 信号置为有效，并返回 wait_brk 状态。

9.4.2　验证电路

我们设计一个简单的串行接口和译码电路来验证 PS2 键盘接口的操作。顶层结构示意图如图 9-5 所示。电路将键的建立码转换为相应的 ASCII 码，并将 ASCII 码送到 UART。相应的字符和数字可以在超级终端窗口中显示。转换电路的 HDL 代码如示例 9.4 所示。

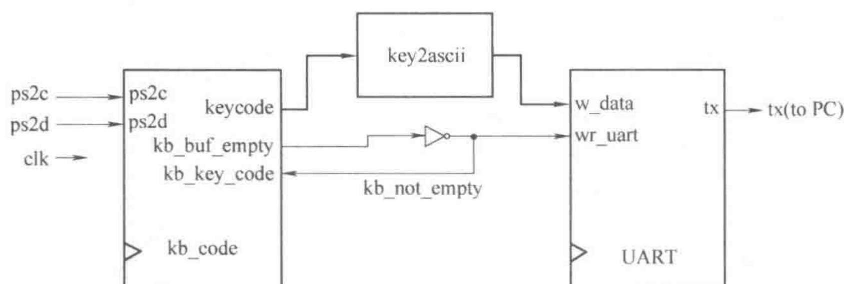


图 9-5　键盘验证电路的框图

示例 9.4　键盘的建立码对应的 ASCII 码

```
module key2ascii
 (
   input wire [7:0] key_code,
  output reg [7: 0] ascii_code
 );
 always@ *
 case(key_code)
    8'h45: ascii_code =8'h30;    // 0
    8'h16: ascii_code =8'h31;    // 1
    8'h1e: ascii_code =8'h32;    // 2
    8'h26: ascii_code =8'h33;    // 3
    8'h25: ascii_code =8'h34;    // 4
    8'h2e: ascii_code =8'h35;    // 5
    8'h36: ascii_code =8'h36;    // 6
    8'h3d: ascii_code =8'h37;    // 7
    8'h3e: ascii_code =8'h38;    // 8
    8'h46: ascii_code =8'h39;    // 9
```

```
8'h1c: ascii_code = 8'h41;    // A
8'h32: ascii_code = 8'h42;    // B
8'h21: ascii_code = 8'h43;    // C
8'h23: ascii_code = 8'h44;    // D
8'h24: ascii_code = 8'h45;    // E
8'h2b: ascii_code = 8'h46;    // F
8'h34: ascii_code = 8'h47;    // G
8'h33: ascii_code = 8'h48;    // H
8'h43: ascii_code = 8'h49;    // I
8'h3b: ascii_code = 8'h4a;    // J
8'h42: ascii_code = 8'h4b;    // K
8'h4b: ascii_code = 8'h4c;    // L
8'h3a: ascii_code = 8'h4d;    // M
8'h31: ascii_code = 8'h4e;    // N
8'h44: ascii_code = 8'h4f;    // O
8'h4d: ascii_code = 8'h50;    // P
8'h15: ascii_code = 8'h51;    // Q
8'h2d: ascii_code = 8'h52;    // R
8'h1b: ascii_code = 8'h53;    // S
8'h2c: ascii_code = 8'h54;    // T
8'h3c: ascii_code = 8'h55;    // U
8'h2a: ascii_code = 8'h56;    // V
8'h1d: ascii_code = 8'h57;    // W
8'h22: ascii_code = 8'h58;    // X
8'h35: ascii_code = 8'h59;    // Y
8'h1a: ascii_code = 8'h5a;    // Z
8'h0e: ascii_code = 8'h60;    // `
8'h4e: ascii_code = 8'h2d;    // -
8'h55: ascii_code = 8'h3d;    // =
8'h54: ascii_code = 8'h5b;    // [
8'h5b: ascii_code = 8'h5d;    // ]
8'h5d: ascii_code = 8'h5c;    // \
8'h4c: ascii_code = 8'h3b;    // ;
8'h52: ascii_code = 8'h27;    // '
8'h41: ascii_code = 8'h2c;    // ,
```

```
      8'h49: ascii_code = 8'h2e;    //.
      8'h4a: ascii_code = 8'h2f;    ///
      8'h29: ascii_code = 8'h20;    // (space)
      8'h5a: ascii_code = 8'h0d;    // (enter, cr)
      8'h66: ascii_code = 8'h08;    // (backspace)
      default: ascii_code = 8'h2a; // *
   endcase
endmodule
```

验证电路的全部代码遵循图 9-5 中的结构示意图，它的代码如示例 9.5 所示。

示例 9.5　键盘验证电路

```
module kb_test
  (
    input wire clk, reset,
    input wire ps2d, ps2c,
    output wire tx
  );
  // 信号声明
  wire [7:0] key_code, ascii_code;
  wire kb_not_empty, kb_buf_empty;
  // 实体
  // 实例化的键盘扫描码电路
  kb_code kb_code_unit
      (.clk(clk), .reset(reset), .ps2d(ps2d), .ps2c(ps2c),
      .rd_key_code(kb_not_empty), .key_code(key_code),
      .kb_buf_empty(kb_buf_empty));
  //UART 实例
  uart uart_unit
      (.clk(clk), .reset(reset), .rd_uart(1'b0),
      .wr_uart(kb_not_empty), .rx(1'b1), .w_data(ascii_code),
      .tx_full(), .rx_empty(), .r_data(), .tx(tx));
  // 键到ASCII 码转换电路实例
  key2ascii k2a_unit
      (.key_code(key_code), .ascii_code(ascii_code));
```

```
  assign kb_not_empty = ~kb_buf_empty;
endmodule
```

9.5 文献备注

Adam Chapweske 写的《PS/2 鼠标/键盘协议》,《PS/2 键盘接口》和《PS/2 鼠标接口》三篇文章，提供了键盘和鼠标接口的详细内容。这些内容可以在 http://www.computer-engineering.org 网站找到。数字系统的高速原型：在 PS2 端口与键盘和鼠标协议上，James 0. Hamblen 的《Quartus@ II Edition》也包含了一章。

9.6 实验

9.6.1 可选的键盘接口 I

在 9.4 节中的接口电路返回最后一个释放键的建立码，因而忽略了连续键入情况。可选的方法是考虑连续键入情况。当一个键保持按下状态时，键盘接口电路应该重复返回按键的建立码并忽略最后的停止码。为了简单，我们假设没有使用扩展键。设计新的接口电路，重新综合验证电路，并检验新接口电路的操作。

9.6.2 可选的键盘接口 II

我们可以扩展接口电路，使它可以分辨"Shift"键是否按下，从而使小写和大写字母都可以键入。扩展电路可进行如下改进：

- 输出键码可以从 8 位扩展到 9 位。额外的位表示移位键是否保持按下；
- FSM 应增加一个专用的分支用来处理"Shift"键的建立和停止码，并据此设置相应位的值；
- FIFO 缓存器的宽度应扩展到 9 位。

设计扩展接口电路，改进 key2ascii 电路来处理小写和大写字母，重新综合验证电路，并检验扩展接口电路的操作。

9.6.3 带看门狗定时器的 PS2 接收子系统

9.2 节中的 PS2 的接收子系统没有容错能力。ps2c 信号的潜在噪声和干扰脉冲可能会引起 FSMD 进入错误的状态。解决问题的一个方法是添加看门狗定时器。每次信号 fall_edge_tick 在状态 get_bit 中被置为有效时，初始化定时器。如

果在下一个 20μs 内没有新的下降沿到达，信号 time_out 被置为有效，并且 FSMD 回到 idle 状态。设计改进的接收子系统，生成一个测试平台，并使用仿真检验它的操作。

9.6.4　键盘控制的秒表

思考实验 4.7.6 中的增强型秒表。在原型板上有 3 个开关控制秒表的操作。我们可以使用键盘向秒表发送命令：

- 当 C 键（表示“清除”）被按下，秒表中断现有的操作，清零，并将计数器方向设置为“向上”；
- 当 G 键（表示“运行”）被按下，秒表开始计数；
- 当 P 键（表示“暂停”）被按下，停止计数；
- 当 U 键（表示“向下”）被按下，秒表反方向计数；
- 忽略所用其他按键。

设计新码表，综合电路并检验它的操作。

9.6.5　键盘控制的移动 LED 横幅

参考实验 4.7.5 中的移动 LED 横幅电路。我们可以使用键盘来控制它的操作并动态的更改横幅中的数字。

- 当 G 键（表示“运行”）按下的时候，LED 横幅开始移动；
- 当 P 键（表示“停止”）按下的时候，LED 横幅停止移动；
- 当 D 键（表示“方向”）按下的时候，LED 横幅反向移动；
- 当一个十进制的数字键（例如，0，1，…，9）按下的时候，横幅会被改变，横幅可以被看为是一个 10 字的 FIFO 缓存器，新的数字会插入到横幅的最前面，并且移出丢弃最左边的数字；
- 忽略所有其他的键。

设计新的移动 LED 横幅电路，综合电路，并检验它的操作。

第 10 章　PS2 鼠标

10.1　引言

计算机的鼠标主要被设计用于在平面上探测二维空间中的运动。它内部的电路测量运动的相对距离和检查按钮的状态。对于一个 PS2 接口的鼠标，这些信息被封装在 3 个数据包里，然后通过 PS2 端口发送到主机。在流模式下，一个 PS2 接口的鼠标可以在预先设定的采样速率下连续地发送包。

PS2 端口的通信是双向的，主机能够发送命令到键盘或鼠标去设置某些参数。从我们的角度出发，这些功能对键盘来说几乎是不需要的，在第 9 章中的这种键盘接口只限于从键盘到 FPGA 主机一个方向的通信。然而，不像键盘，鼠标在上电后被设置在非连续模式下并且没有发送任何数据。主机首先要发送一条命令给鼠标去初始化鼠标并使能连续模式。因此，PS2 端口的双向通信是 PS2 鼠标接口所需要的，我们必须为 PS2 接口设计一个传输子系统(从 FPGA 板到鼠标)。

在本章，我们会提供一个 PS2 鼠标协议的简略总结，设计一个双向的 PS 接口，并得到一个简单的鼠标接口。

10.2　PS2 鼠标协议

10.2.1　基本操作

标准的 PS2 鼠标会报告 x 轴(右/左)、y 轴(上/下)的运动和左侧按钮、中间按钮、右侧按钮的状态。每次移动量的大小被记录在鼠标内部的计数器中。在数据发送到主机的时候，计数器就会清零且重新开始计算。计数器的内容用一个 9 位的有符号整型表示，正数表示向右或向上的运动，负数表示向左或向下的运动。

物理距离之间的关系是通过鼠标的灵敏度参数来定义。灵敏度默认值是 4 个计数每毫米。在鼠标连续地运动的时候，数据在规定的速率下发送。速率是通过鼠标采样速率参数来定义。采样速率的默认值是每秒采样 100。如果鼠标运动速度太快，在采样周期期间移动量的大小可能超过计数器的最大范围。计数器在适

当的方向上设置成最大值。两个溢出位标识了这种情况。

通过 3 字节数据鼠标能报告它的运动和按钮的行为，它嵌入在 3 个 PS2 数据包中。3 字节数据的详细格式在表 10-1 中进行说明。它包含以下信息：

- x_8，…，x_0：用二进制补码表示沿 x 轴的运动；
- x_v：x 轴运动的溢出值；
- y_8，…，y_0：用二进制补码表示沿 y 轴的运动；
- y_v：y 轴运动的溢出值；
- l：左侧按钮的状态，在左侧按钮进行按压的时候它是 1；
- r：右侧按钮的状态，在右侧按钮进行按压的时候它是 1；
- m：可选择的中间按钮的状态，在中间按钮进行按压的时候它是 1。

在发送数据期间，首先发送字节数据包 1，最后发送字节数据包 3。

表 10-1　鼠标数据包格式

byte1	y_v	x_v	y_8	x_8	1	m	r	l
byte2	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
byte3	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

10.2.2　基本的初始化程序

鼠标的操作比键盘复杂得多。它有不同的操作模式。最常用的就是流模式，在鼠标探测到运动或按钮行为的时候，它会发送数据。如果运动是连续的，数据会在指定的采样速率下产生。

在操作期间，主机可以发送命令给鼠标去修改不同参数的默认值并设置操作模式，鼠标也能生成状态数据并发送一个应答。对于我们而言，默认值已经足够了，只要将鼠标设置在流模式下就可以了。

一个 PS2 鼠标和 FPGA 主机之间基本的交互顺序由以下部分组成：

1）上电时，鼠标完成内部的上电测试，鼠标发送 1 字节数据 AA，它显示测试通过，然后发送 1 字节数据 00，这是一个标准 PS2 鼠标的 id 号；

2）FPGA 主机发送命令 F4，去使能流模式，鼠标将用 FE 进行响应去确认这条命令接收到。

3）现在鼠标进入流模式并发送标准的数据包。

如果鼠标预先已经插到 FPGA 板子上，它在板子上电的时候立即接通并发送 AA 00 数据以完成上电测试。在该时刻 FPGA 芯片没有配置并且不会接收数据。因此，我们通常可以忽略上电后第一阶段的通信。最小的鼠标接口电路只需要发送 F4 命令，检查 FE 应答，并进入标准的操作模式去处理鼠标规则的数据包。

我们可以通过复位命令去强制鼠标返回到初始状态：

1）FPGA 主机发送命令 FF 去复位鼠标，鼠标将用 FE 进行响应去确认这条命令接收到；

2）鼠标完成内部的上电测试并发送 AA 00，在程序运行过程中流模式将会禁止。

最新的鼠标增加了更多的功能性，例如一个滚轮和额外的按钮，从而发送更多的信息。额外的字节会添加到最初的 3 字节数据中去适应这些新的特性。

10.3 PS2 传输子系统

10.3.1 主机到 PS2 设备的通信协议

主机到 PS2 设备的通信协议包括双向数据交换。实际上鼠标数据和时钟线是 OC 门电路。对于我们的设计目的，我们把它们当作是三态。从主机到 PS2 设备传输数据包的基本时序图在图 10-1 中进行说明，在该图中数据和时钟信号的标识是 ps2d 和 ps2c。为了清晰的描述，将该图分成两个部分进行描述，一部分行为由主机产生（FPGA 芯片），一部分行为由设备产生（鼠标）。基本操作顺序如下所示：

1）主机强制 ps2c 线至少保持 100μs 的低电平从而禁止任何鼠标的行为，这步可以认为是主机请求发送信息包；

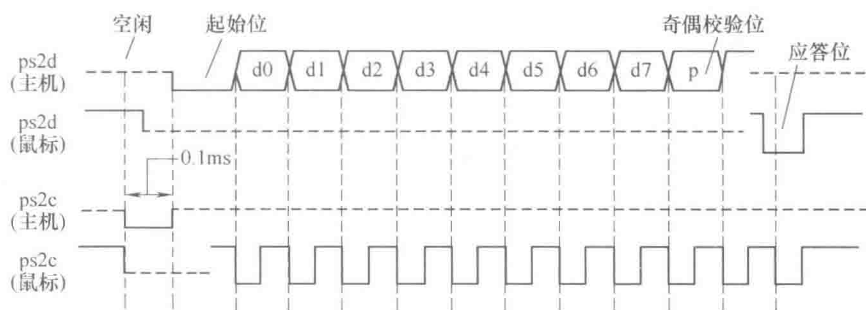


图 10-1 PS2 端口的主机到设备的时序图

2）主机强制 ps2d 线为低电平并禁止 ps2c 线（设为高阻），该步认为是主机发送的一个起始位；

3）现在 PS2 设备接收 ps2c 线而且负责产生 PS2 的时钟信号，当检测到起始位后，PS2 设备会产生 1 到 0 的变化；

4）一旦检测到这个变化，主机在 ps2d 线上输出一位有效数据，它会保持这

个有效值直到在 ps2c 线上 PS2 设备产生 1 到 0 的变化，这在本质上是对接收到数据的确认；

5）为了传递其余 7 个数据位和 1 个奇偶校验位重复第 4 步；

6）发送完奇偶校验位后，主机会禁止 ps2d 线（设为高阻），现在 PS2 设备接收 ps2d 线并通过将 ps2d 线置 0 的方式确认完成通信。按照要求，主机可以在 ps2c 线中最后 1 到 0 变化时检查这些值去确认数据包已经成功地发送。

10.3.2　设计和代码

不像接收子系统，ps2c 和 ps2d 信号是双向的通信。三态缓存对于每个信号是必需的。三态接口在图 10-2 中已进行描述。tri_c 和 tri_d 信号是使能信号，该信号可以控制三态缓存。当它们有效时，相应的 ps2c_out 和 ps2d_out 信号将会发送到输出端口。

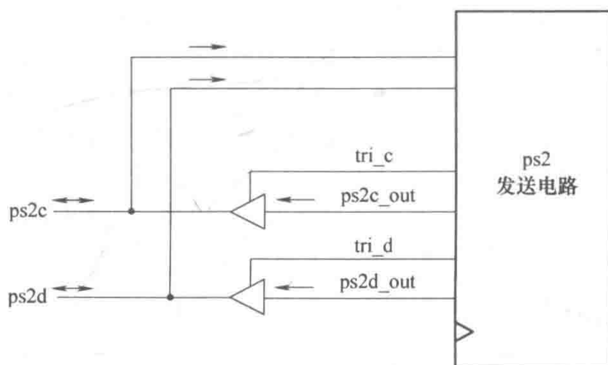


图 10-2　PS2 传输子系统的三态缓存

当设计发送子系统时，我们可以按照前面协议的顺序去设计一个 ASMD 图，它在图 10-3 中进行说明。FSMD 初始处在空闲状态。开始发送时，主机置位 wr_ps2 信号并在总线上放置数据。FSMD 写数据及奇偶校验位等到移位寄存器，写“1…1”到 c_reg，并跳到 rst 状态（“请求发送”状态）。在这个状态，将 ps2c_out 置为 0，且相应的 tri_c 信号置为有效去使能相应的三态缓存。c_reg 常用 13bit 计数器去产生 164μs 延时。于是 FSMD 跳到 start 状态，在这个状态 PS2 时钟线是禁止的并将数据线设为 1。现在 PS2 设备（鼠标）在 ps2c 线上接收并产生时钟信号。在检测到 ps2c 信号的下降沿后，通过下降沿信号，FSMD 跳到 data 状态并移动 8 个数据位和一个奇偶校验位。n 寄存器用于跟踪记录比特移动的计数。然后 FSMD 跳到 stop 状态，在这个状态数据线是禁止的。当检测到最后一个下降沿后 FSMD 返回到空闲状态。

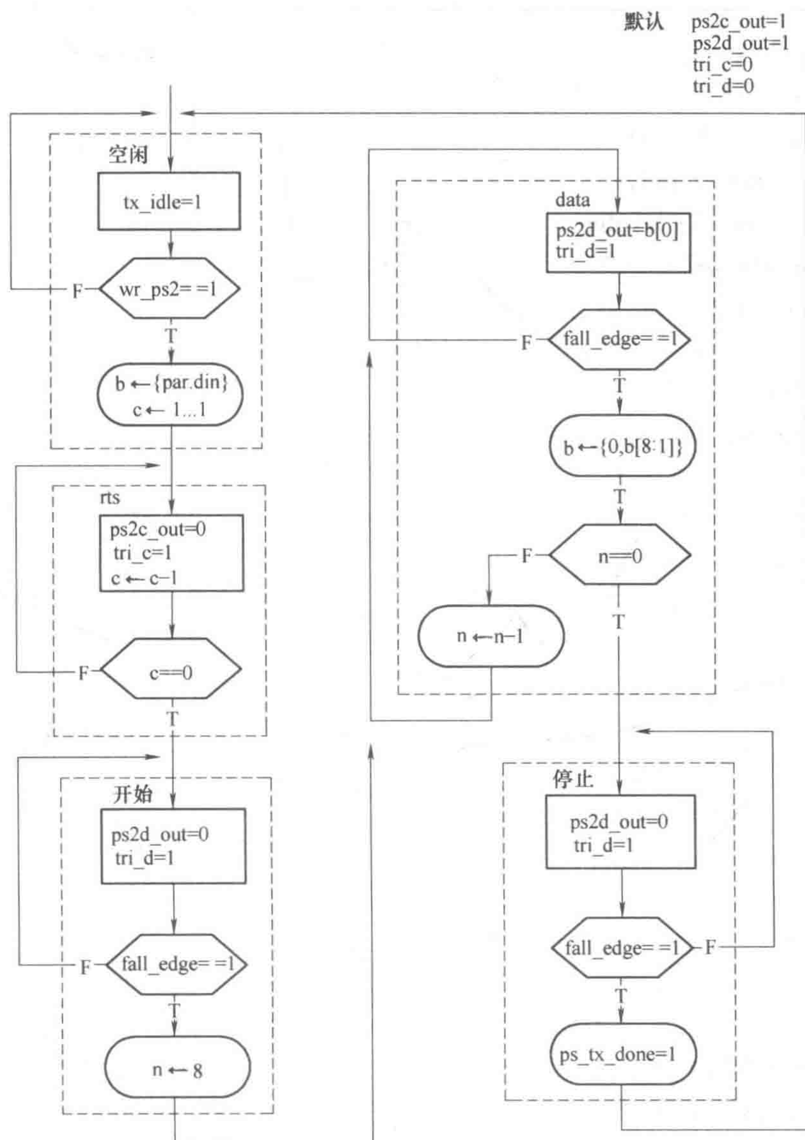


图 10-3　PS2 发送子系统的 ASMD 图

FSMD 也包括 tx_idle 信号标示是否处在发送的过程中。这个信号用于协调发送和接受子系统之间的操作。示例 10.1 中列出了 ASMD 图的代码。滤波器电路和 9.2 节类似用于产生下降沿信号。

示例 10.1　PS2 端口发送器

```
module ps2_tx
  (
    input wire clk, reset,
    input wire wr_ps2,
    input wire [7:0] din,
    inout wire ps2d, ps2c,
    output reg tx_idle, tx_done_tick
  );
  // 字符状态声明
localparam [2:0]
       idle   =3'b000,
       rts    =3'b001,
       start =3'b010,
       data   =3'b011,
       stop   =3'b100;
  // 信号声明
  reg [2:0] state_reg, state_next;
  reg [7:0] filter_reg;
  wire [7:0] filter_next;
  reg f_ps2c_reg;
  wire f_ps2c_next;
  reg [3:0] n_reg, n_next;
  reg [8:0] b_reg, b_next;
  reg [12:0] c_reg, c_next;
  wire par, fall_edge;
  reg ps2c_out, ps2d_out;
  reg tri_c, tri_d;
  // 实体
  // ===============================================
  // 产生滤波器和下降沿记号
  // ===============================================
  always @(posedge clk, posedge reset)
  if (reset)
```

```
        begin
          filter_reg  <=0;
          f_ps2c_reg  <=0;
        end
    else
        begin
          filter_reg  <=filter_next;
          f_ps2c_reg  <=f_ps2c_next;
        end
    assign filter_next = {ps2c, filter_reg[7: 1]};
    assign f_ps2c_next = (filter_reg ==8'b11111111) ? 1'b1 :
                         (filter_reg ==8'b00000000) ? 1'b0 :
                         f_ps2c_reg;
    assign fall_edge = f_ps2c_reg & ~f_ps2c_next;
    // ==================================================
    //FSMD
    // ==================================================
    //FSMD 状态和数据寄存器
always @ (posedge clk, posedge reset)
        if (reset)
          begin
              state_reg  <= idle;
              c_reg  <=0;
              n_reg  <=0;
              b_reg  <=0;
          end
        else
          begin
              state_reg  <= state_next;
              c_reg  <= c_next;
              n_reg  <= n_next;
              b_reg  <= b_next;
          end
    // 奇偶校验位
    assign par = ~ (^din);
```

```
  //FSMD 次态逻辑
always @ *
  begin
      state_next = state_reg;
      c_next = c_reg;
      n_next = n_reg;
      b_next = b_reg;
      tx_done_tick = 1'b0;
      ps2c_out = 1'b1;
      ps2d_out = 1'b1;
      tri_c = 1'b0;
      tri_d = 1'b0;
      tx_idle = 1'b0;
      case (state_reg)
        idle:
            begin
              tx_idle = 1'b1;
              if (wr_ps2)
                  begin
                    b_next = {par, din};
                    c_next = 13'h1fff; //2^13 - 1
                    state_next = rts;
                  end
            end
        rts: // 请求发送
            begin
              ps2c_out = 1'b0;
              tri_c = 1'b1;
              c_next = c_reg - 1;
              if (c_reg == 0)
                  state_next = start;
            end
        start: // 起始位有效
            begin
              ps2d_out = 1'b0;
```

```
            tri_d = 1'b1;
            if (fall_edge)
                begin
                    n_next = 4'h8;
                    state_next = data;
                end
        end
    data: //8 个数据加一个奇偶校验位
        begin
            ps2d_out = b_reg[0];
            tri_d = 1'b1;
            if (fall_edge)
                begin
                    b_next = {1'b0, b_reg[8:1]};
                    if (n_reg == 0)
                        state_next = stop;
                    else
                        n_next = n_reg - 1;
                end
        end
    stop: // 假设ps2d 变高
        if (fall_edge)
            begin
                state_next = idle;
                tx_done_tick = 1'b1;
            end
  endcase
end
// 三态缓存
assign ps2c = (tri_c) ? ps2c_out : 1'bz;
assign ps2d = (tri_d) ? ps2d_out : 1'bz;
endmodule
```

该代码中没有错误检测电路。更加健壮的设计应该检查奇偶校验的正确性和确认比特位并包括看门狗定时器以防止鼠标在不正确的状态被锁定。

10.4　双向的 PS2 接口

10.4.1　基本设计和代码

我们可以结合接收和发送子系统形成双向的 PS2 接口。在图 10-4 中描述了顶层模块图。我们用 tx_idle 和 rx_en 信号去协调发送和接收操作。发送操作具有优先权。当发送子系统正在工作时，tx_idle 信号无效，相应的禁止接收子系统。只有当发送子系统在空闲状态时接收子系统才能处理输入。示例 10.2 中列出了相应的 HDL 代码。

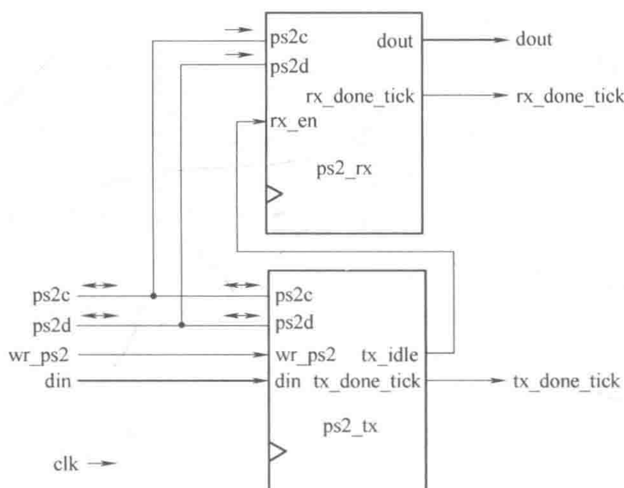


图 10-4　双向 PS2 接口的顶层模块图

示例 10.2　双向的 PS2 接口

```
module ps2_rxtx
  (
    input wire clk, reset,
    input wire wr_ps2,
    inout wire ps2d, ps2c,
    input wire [7: 0] din,
    output wire rx_done_tick, tx_done_tick,
    output wire [7: 0]dout
  );
```

```
// 信号声明
wire tx_idle;
// 主体
//ps2 接收实例
ps2_rx ps2_rx_unit
    (.clk(clk), .reset(reset), .rx_en(tx_idle),
    .ps2d(ps2d), .ps2c(ps2c),
    .rx_done_tick(rx_done_tick), .dout(dout));
//ps2 发送实例
ps2_tx ps2_tx_unit
    (.clk(clk), .reset(reset), .wr_ps2(wr_ps2),
    .din(din), .ps2d(ps2d), .ps2c(ps2c),
    .tx_idle(tx_idle), .tx_done_tick(tx_done_tick));
endmodule
```


```
// 信号声明
wire tx_idle;
// 主体
//ps2 接收实例
ps2_rx ps2_rx_unit
  (.clk(clk), .reset(reset), .rx_en(tx_idle),
    .ps2d(ps2d), .ps2c(ps2c),
    .rx_done_tick(rx_done_tick), .dout(dout));
//ps2 发送实例
ps2_tx ps2_tx_unit
  (.clk(clk), .reset(reset), .wr_ps2(wr_ps2),
    .din(din), .ps2d(ps2d), .ps2c(ps2c),
    .tx_idle(tx_idle), .tx_done_tick(tx_done_tick));
endmodule
```


10.4.2　确认电路

我们设计一个测试电路去检验和监控双向的接口操作。在图 10-5 中描述了模块图。命令是手动发送的。我们使用 8bit 开关设置数据（主机的命令）并使用按钮产生一个时钟周期记号去发送信息包。接收的信息包数据开始通过字节到 ASCII 转换电路，这是转换数据到两个 ASCII 字符加上一个空格。字符然后通过

UART 传送并显示在 Windows 终端上。示例 10.3 中列出了 HDL 代码。



图 10-5 鼠标监控电路模块图

示例 10.3 双向的 PS2 接口监控电路

```
module ps2_monitor
  (
    input wire clk, reset,
    input wire [7: 0] sw,
    input wire [2: 0]btn,
    inout wire ps2d, ps2c,
    output wire tx
  );
  // 常量声明
  localparam SP = 8'h20; //ASCII 中的空格
  // 字符状态声明
  localparam [1: 0]
      idle   =2'b00,
      send1 =2'b01,
      send0 =2'b10,
      sendb =2'b11;
  // 信号声明
  reg [1: 0] state_reg, state_next;
  wire [7: 0] rx_data;
  reg [7: 0] w_data, ascii_code;
  wire psrx_done_tick, wr_ps2;
  reg wr_uart;
  wire [3: 0] hex_in;
  // 主体
```

```
// =============================================
// 实例
// =============================================
//ps2 接收器/发送器实例
ps2_rxtx ps2_rxtx_unit
     (.clk(clk), .reset(reset), .wr_ps2(wr_ps2),
     .din(sw), .dout(rx_data), .ps2d(ps2d), .ps2c(ps2c),
     .rx_done_tick(psrx_done_tick), .tx_done_tick());
//UART 实例(只使用UART 接收器)
uart uart_unit
     (.clk(clk), .reset(reset), .rd_uart(1'b0),
     .wr_uart(wr_uart), .rx(1'b1), .w_data(w_data),
     .tx_full(), .rx_empty(), .r_data(), .tx(tx));
// 反跳电路实例
debounce btn_db_unit
     (.clk(clk), .reset(reset), .sw(btn[0]),
     .db_level(), .db_tick(wr_ps2));
// =============================================
// 发送3 个ASCII 字符状态机
// =============================================
// 状态寄存器
always @(posedge clk, posedge reset)
     if (reset)
        state_reg <= idle;
     else
        state_reg <= state_next;
// 次态逻辑
always @ *
begin
     wr_uart = 1'b0;
     w_data = SP;
     state_next = state_reg;
     case (state_reg)
        idle:
             if (psrx_done_tick) // 查看接收代码
```

```
            state_next = send1;
      send1: // 发送高十六进制字符
         begin
            w_data = ascii_code;
            wr_uart = 1'b1;
            state_next = send0;
         end
      send0: // 发送低十六进制字符
         begin
            w_data = ascii_code;
            wr_uart = 1'b1;
            state_next = sendb;
         end
      sendb: // 发送空字符
         begin
            w_data = SP;
            wr_uart = 1'b1;
            state_next = idle;
         end
   endcase
end
// ==========================================
// 查看ASCII 显示器代码
// ==========================================
// 分离4bit 十六进制代码
assign hex_in = (state_reg == send1) ? rx_data[7:4] :
                                       rx_data[3:0];
// 十六进制数转换成ASCII 码
always @ *
   case (hex_in)
      4'h0: ascii_code = 8'h30;
      4'h1: ascii_code = 8'h31;
      4'h2: ascii_code = 8'h32;
      4'h3: ascii_code = 8'h33;
      4'h4: ascii_code = 8'h34;
```

```
    4'h5: ascii_code =8'h35;
    4'h6: ascii_code =8'h36;
    4'h7: ascii_code =8'h37;
    4'h8: ascii_code =8'h38;
    4'h9: ascii_code =8'h39;
    4'ha: ascii_code =8'h41;
    4'hb: ascii_code =8'h42;
    4'hc: ascii_code =8'h43;
    4'hd: ascii_code =8'h44;
    4'he: ascii_code =8'h45;
    default: ascii_code =8'h46;
  endcase
endmodule
```

如果鼠标连接到PS2电路，我们首先发送FF命令去复位鼠标，然后发送F4命令去使能流模式。Windows超级终端将会显示鼠标确认包和后来鼠标运动信息包。

10.5 PS2鼠标接口

10.5.1 基本设计

基本的PS2鼠标接口是在双向PS2电路上创建另一个电路层。它的两个基本功能是启动流模式和重新装载3数据字节。电路的输出由xm和ym组成，这是两个9bit x和y轴运动的信号；btm，这是3bit按钮状态信号；以及m_done_tick，这是一个时钟周期状态信号，当装载的数据有效时置位。

示例10.4中列出了HDL代码。它通过7个状态的FSMD来实现。一旦复位信号有效后完成init1、init2和init3状态。在这些状态中，FSMD发出F4命令，等待发送完成，然后等待确认包。鼠标现在流模式下。然后FSMD得到信息包并在pack1、pack2、pack3状态中重新写入另外的3个信息包，然后在done状态下激活m_done_tick信号。FSMD循环这4个状态。

示例10.4 基本鼠标接口电路

```
module mouse
  (
```

```
  input wire clk,reset,
  inout wire ps2d,ps2c,
  output wire [8:0] xm,ym,
  output wire [2:0]btnm,
  output reg   m_done_tick
);
//常量声明
localparam STRM =8'hf4; //命令F4
//字符状态声明
localparam [2: 0]
     init1 =3'b000,
     init2 =3'b001,
     init3 =3'b010,
     pack1 =3'b011,
     pack2 =3'b100,
     pack3 =3'b101,
     done    =3'b110;
//信号声明
reg [2: 0] state_reg, state_next;
wire [7: 0] rx_data;
reg wr_ps2;
wire rx_done_tick, tx_done_tick;
reg [8: 0] x_reg, y_reg, x_next, y_next;
reg [2: 0]btn_reg, btn_next;
//主体
//实例
ps2_rxtx ps2_unit
     (.clk(clk), .reset(reset), .wr_ps2(wr_ps2),
     .din(STRM), .dout(rx_data), .ps2d(ps2d), .ps2c(ps2c),
     .rx_done_tick(rx_done_tick),
     .tx_done_tick(tx_done_tick));
//主体
//FSM 状态和data 状态
always @ (posedge clk, posedge reset)
     if (reset)
```

```
      begin
          state_reg  <= init1;
          x_reg  <=0;
          y_reg  <=0;
          btn_reg  <=0;
      end
    else
      begin
          state_reg  <= state_next;
          x_reg  <= x_next;
          y_reg  <= y_next;
          btn_reg  <= btn_next;
      end
//FSMD 次态逻辑
always @ *
begin
    state_next = state_reg;
    wr_ps2 = 1'b0;
    m_done_tick = 1'b0;
    x_next = x_reg;
    y_next = y_reg;
    btn_next = btn_reg;
    case (state_reg)
      init1:
          begin
            wr_ps2 = 1'b1;
            state_next = init2;
          end
      init2: // 等待发送完成
          if (tx_done_tick)
            state_next = init3;
      init3: // 等待确认包
          if (rx_done_tick)
            state_next = pack1;
      pack1: // 等待第一个数据包
```

```
            if (rx_done_tick)
               begin
                    state_next = pack2;
                    y_next[8] = rx_data[5];
                    x_next[8] = rx_data[4];
                    btn_next =    rx_data[2:0];
               end
        pack2: // 等待第二个数据包
            if (rx_done_tick)
               begin
                    state_next = pack3;
                    x_next[7:0] = rx_data;
               end
        pack3: // 等待第三个数据包
            if (rx_done_tick)
               begin
                    state_next = done;
                    y_next[7:0] = rx_data;
               end
        done:
            begin
               m_done_tick = 1'b1;
               state_next = pack1;
            end
     endcase
  end
  // 输出
  assign xm = x_reg;
  assign ym = y_reg;
  assign btnm = btn_reg;
endmodule
```

该设计只提供了最基本的功能项。类似于 8.2.4 节，更加高级的电路应有更健壮的方法去启动流模式并增加一个额外的缓冲区，实现与外部系统更好的交互。

10.5.2　测试电路

我们用简单的测试电路去验证 PS2 接口。本电路使用鼠标去控制 8 个板子上的 LED 灯。只有其中的一个 LED 灯亮且亮灯的位置跟着 x 轴鼠标移动。按鼠标左键和右键分别点亮在最左边的位置或最右边的位置上的 LED 灯。

示例 10.5 中列出了 HDL 代码。它使用了一个 10 位计数器跟踪记录当前 x 轴的位置。当新的数据项有效时该计数器更新。当按下左边或右边鼠标按钮时，计数器重置为 0 或最大值。否则，其加上这个沿着 x 轴移动量的有符号数值。译码电路用计数器的 3 个最高有效位去激活其中的一个 LED 灯。

示例 10.5　鼠标控制 LED 电路

```
module mouse_led
  (
    input wire clk, reset,
    inout wire ps2d, ps2c,
    output reg [7: 0] led
  );
  // 信号声明
  reg [9: 0] p_reg;
  wire [9: 0] p_next;
  wire [8: 0] xm;
  wire [2: 0]btnm;
  wire m_done_tick;
  // 主体
  // 实例
  mouse mouse_unit
      (.clk(clk), .reset(reset), .ps2d(ps2d), .ps2c(ps2c),
      .xm(xm), .ym(), .btnm(btnm),
      .m_done_tick(m_done_tick));
  // 计数器
  always @ (posedge clk, posedge reset)
      if (reset)
        p_reg <=0;
      else
        p_reg <=p_next;
```

```
  assign p_next = ( ~m_done_tick) ? p_reg   : // 未激活
                  (btnm[0])        ? 10'b0   : // 左边按钮
                  (btnm[1])        ? 10'h3ff : // 右边按钮
                  p_reg + {xm[8], xm};         //x 轴运动
  always @ *
     case (p_reg[9:7])
       3'b000: led = 8'b10000000;
       3'b001: led = 8'b01000000;
       3'b010: led = 8'b00100000;
       3'b011: led = 8'b00010000;
       3'b100: led = 8'b00001000;
       3'b101: led = 8'b00000100;
       3'b110: led = 8'b00000010;
       default: led = 8'b00000001;
     endcase
endmodule
```

10.6 文献备注

本章附录资料与第 9 章类似。

10.7 实验

鼠标主要用于图像视频接口，这在 13 章和 14 章进行了描述。许多附加的相关鼠标实验能够在这些章节中找到。

10.7.1 键盘控制电路

主机也能够发送命令对 PS2 键盘设置某个参数。例如，我们能够通过发送 ED0X 命令控制键盘的 3 个 LED 灯。这个 X 用“0snc”的格式表示十六进制数，其中 s、n 和 c 是 1bit 值，它是分别控制滚动锁定、数码锁定和大写锁定的 LED 灯。我们可以将这些特征合并到 9.4.1 节键盘接口电路中，并用 3bit 开关去控制这 3 个键盘的 LED 灯。设计扩展接口电路、重新综合电路和验证操作。

10.7.2 增强的鼠标接口

鼠标接口在 10.5 节已描述，我们可以通过手动使能或禁止流模式的方式改变设计。这可以用 FPGA 板的两个按钮来实现。一个按钮发送复位命令 FF，它可以在操作过程中禁止流模式，而另一个按钮发送 F4 命令去使能流模式。我们可以修改原始接口去合并这些特征，重新综合 LED 测试电路去验证这些操作。

10.7.3 鼠标控制 7 段 LED 显示器

我们可以用鼠标在四位数的七段 LED 显示器中输入 4 位十进制数。该电路的功能如下：

- 只有 LED 显示器 4 个小数点中的一个是亮的，这个亮灯的小数点说明选择的数字位置；
- 选择的数字位置跟着鼠标的 x 轴移动；
- 选择的 7 段 LED 显示器内容是十进制数(0，…，9)和鼠标 y 轴移动的变化。

设计和综合这个电路并验证其操作。

第11章　外部SRAM

11.1　引言

随机存取存储器（RAM）在数字系统中被作为大容量存储器使用，因为一个RAM的单元比一个 触发器单元简单许多。RAM的一般使用类型是异步静态存储器(SRAM)。寄存器的工作原理是在时钟信号的一个上升沿或下降沿到来时对数据进行采样和存储，而从异步SRAM中访问数据比之更加复杂。读写操作要求数据、地址和控制信号在特定时序下有效，并且上述信号在进行读写操作的这一段时间内必须是稳定的状态。

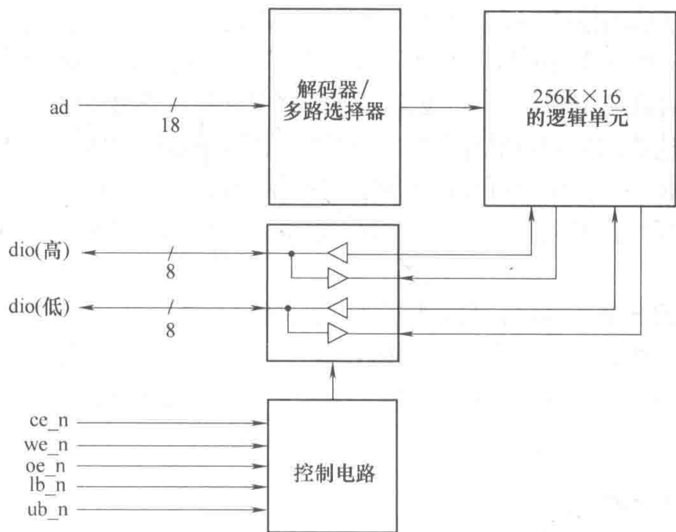
同步系统直接访问SRAM难以实现。我们通常采用存储控制器作为外部接口，外部接口同步地从主系统接收命令，然后产生适当的同步信号去访问SRAM。控制器通过详细的时序要求保护主系统，并使得存储器访问就像同步操作。存储控制器的性能是通过在给定的周期内完成访问寄存器的次数来衡量。虽然设计一个简易的存储控制器是简单的，但是处理好时序问题以达到最佳性能要求是十分困难的。

S3板有2个256K×16异步SRAM器件，即共有1M字节的存储空间。在本章，我们将展示这些器件的存储控制器结构。因为每个RAM器件的时序特性是不同的，控制器只适合于特定的器件。然而，相同的设计原理可以应用于类似的SRAM器件。Xilinx Spartan-3器件也包含少量的嵌入存储块。这部分存储器的使用在第12章论述。

11.2　IS61LV25616AL SRAM的特性

11.2.1　Block示意图和I/O信号

S3板有两个IS61LV25616AL型器件，这些器件是集成硅解决方案公司(Integrated Silicon Solution，Inc.(ISSI))制造的256K×16SRAM。简化的Block示意图如图11-1a所示。这个器件具有18bit地址总线ad、双向的16bit数据总线dio和5组控制信号。数据总线分为高字节和低字节，高、低字节可单独访问。5组控制信号如下：



a) Block示意图

操作	ce_n	we_n	oe_n	lb_n	ub_n	dio(低)	dio(高)
无效	1	-	-	-	-	Z	Z
	0	1	1	-	-	Z	Z
	0	-	-	1	1	Z	Z
读	0	1	0	0	1	数据输出	Z
	0	1	0	1	0	Z	数据输出
	0	1	0	0	0	数据输出	数据输出
写	0	0	-	0	1	数据写入	Z
	0	0	-	1	0	Z	数据写入
	0	0	-	0	0	数据写入	数据写入

b) 功能表

操作	we_n	oe_n	dio(16bits)
输出无效	1	1	Z
读16bit字	1	0	数据输出
写16bit字	0	-	数据写入

c) 简化功能表

图 11-1　ISSI 256K×16 SRAM 的 BLOCK 示意图和功能

- ce_n(片选使能)：芯片有效或无效控制信号；
- we_n(写使能)：写操作有效或无效控制信号；
- oe_n(输出使能)：输出有效或无效控制信号；
- lb_n(低字节使能)：数据总线的低字节有效或无效控制信号；
- ub_n(高字节使能)：数据总线的高字节有效或无效控制信号。

所有的这些信号均是低有效，并且用 n 后缀强调这个特性。具体的功能表如图 11-1b 所示。ce_n 信号可以用来调整存储器的扩展，we_n 信号和 oe_n 信号分别用来控制写操作和读操作。lb_n 信号和 ub_n 信号用于高低字节(byte-oriented)配置。

在本章的剩余内容中，我们将举例说明存储控制器的设计和时序问题。为了更加清晰的阐明，我们使用单个 SRAM 器件，以 16bit 字的形式访问 SRAM。这就意味着 ce_n、lb_n 和 ub_n 信号应一直有效(即，约束为 0)。简化功能表如图 11-1c 所示。

11.2.2　时序参数

异步 SRAM 的时序特性十分复杂，包含的参数超过 24 个。我们仅专注于与我们设计相关的几个关键参数。

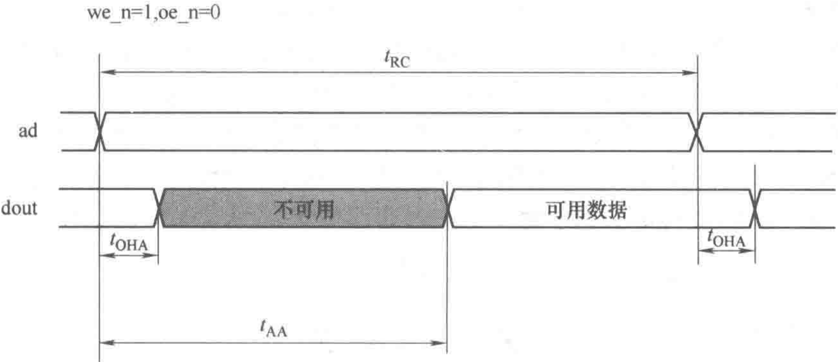
两种类型的读操作简化时序图如图 11-2a 和图 11-2b 所示。相应的时序参数有：

- t_{RC}：读周期时间，两次读操作之间经过的最短时间，对于 SRAM 来说，该参数与 t_{AA}类似；
- t_{AA}：地址访问时间，从地址变换之后到获得稳定的输出数据所需要的时间；
- t_{OHA}：输出数据保持时间，从地址变换到数据信号无效的时间，不要将上述时间与边沿触发 FF 的保持时间混淆，边沿触发 FF 是输入接口 d 的约束；
- t_{DOE}：输出使能访问时间，从 oe_n 有效之后到获得有效数据所需要的时间；
- t_{HZOE}：输出使能到 high-Z 的时间，从 oe_n 无效之后到数据总线为高阻的时间；
- t_{LZOE}：输出使能到 low-Z 的时间，从 oe_n 有效之后到数据总线离开高阻的时间。需要注意的是，尽管此时输出已经不是在高阻态，数据依然是无效的。

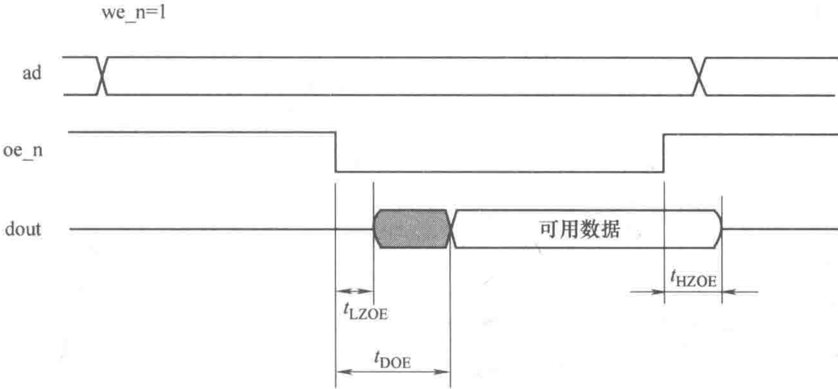
IS61LV25616AL 器件中这些参数的值如图 11-2c 所示。

由 we_n 控制的写操作简化时序示意图如图 11-3a 所示。相应的时序参数如下：

- t_{WC}：写周期时间，两次写操作之间的最短时间；



a) 由地址控制的一个周期的读操作的时序示意图



b) 由oe_n控制的一个周期的读操作的时序示意图

参数		最小	最大
t_{RC}	读周期时间	10	–
t_{AA}	地址访问时间	–	10
t_{OHA}	输出数据保持时间	2	–
t_{DOE}	输出使能访问时间	–	4
t_{HZOE}	输出使能到high-Z的时间	–	4
t_{LZOE}	输出使能到low-Z的时间	0	–

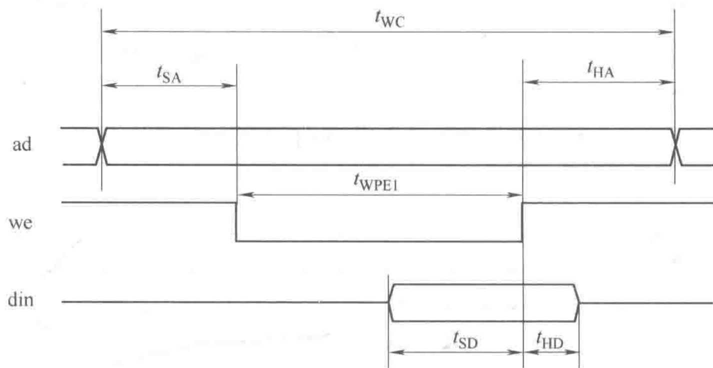
c) 时序参数(单位：ns)

图 11-2　读操作的时序示意图和参数

- t_{SA}：地址建立时间，从 we_n 有效之后到地址稳定的最短时间；
- t_{HA}：地址保持时间，从 we_n 无效之后到地址稳定的最短时间；

- t_{PWE1}：we_n 脉冲宽度，we_n 必须有效的最短时间；
- t_{SD}：数据的建立时间，从数据稳定之后到 we_n 由 0 变为 1 的跳变沿(该跳变沿也称为 THE LATCHING EDGE)的最短时间；
- t_{HD}：数据保持时间，从 we_n 由 0 变为 1 之后到数据无效的最短时间。

IS61LV25616AL 上述参数的值如图 11-3b 所示。全部的时序信息可参见 IS61LV25616AL 器件的数据手册。



a) 写周期的时序

参数		最小	最大
t_{WC}	写周期时间	10	–
t_{SA}	地址建立时间	0	–
t_{HA}	地址保持时间	0	–
t_{PWE1}	脉冲宽度	8	–
t_{SD}	数据的建立时间	6	–
t_{HD}	数据保持时间	0	–

b) 时序参数(单位：ns)

图 11-3　写操作的时序示意图和参数

11.3　基础存储控制器

11.3.1　Block 示意图

存储控制器的连接关系示意图及输入/输出信号如图 11-4 所示。图中 SRAM

这边的信号会在11.2.1节论述。在主系统这边的信号有：

● mem：当该信号为1时，开始存储器的读写操作；

● r/w：详细说明当前操作是读操作(1)还是写操作(0)；

● addr：18bit的地址总线；

● data_f2s：写入SRAM的16bit数据(后缀_f2s代表着FPGA to SRAM)；

● data_s2f_r：从SRAM中重新获得的16bit寄存器数据(后缀_s2f代表着SRAM to FPGA)；

● data_s2f_ur：从SRAM中重新获得的16bit非寄存数据；

● ready：该信号是状况信号，标志着控制器已准备好接收新的命令。当存储器的操作时间超过一个时钟周期时，需要有ready信号。

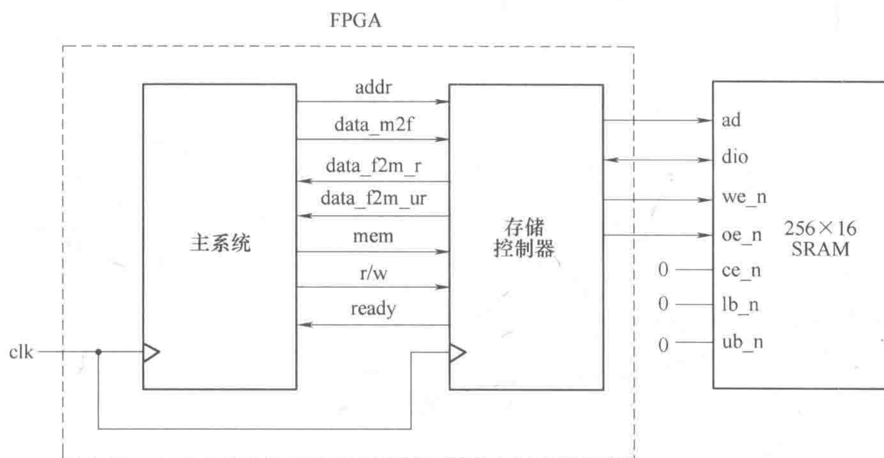


图11-4　SRAM存储控制器的连接关系示意图

存储控制器主要是在SRAM中提供“同步约束”(synchronous wrap)。当主系统想访问存储器时，它将地址和数据信号(对写操作)放在总线上，并激活命令(即：mem和rw信号)。在时钟的上升沿，所有的信号被存储控制器采样，从而执行想要的操作。对于读操作来说，数据信号在一到两个时钟周期之后变成是可访问的。

存储控制器的block示意图如图11-5所示。它的数据通道包含一个存储地址的地址寄存器和两个分别存储各自用途的数据寄存器。由于数据总线(data bus)和dio是双向信号，我们使用三态缓冲器对其进行控制。本设计中使用一个状态机(FSM)产生一个合适的控制时序，时序关系参考图11-2和图11-3中的时序示意图和规格说明。

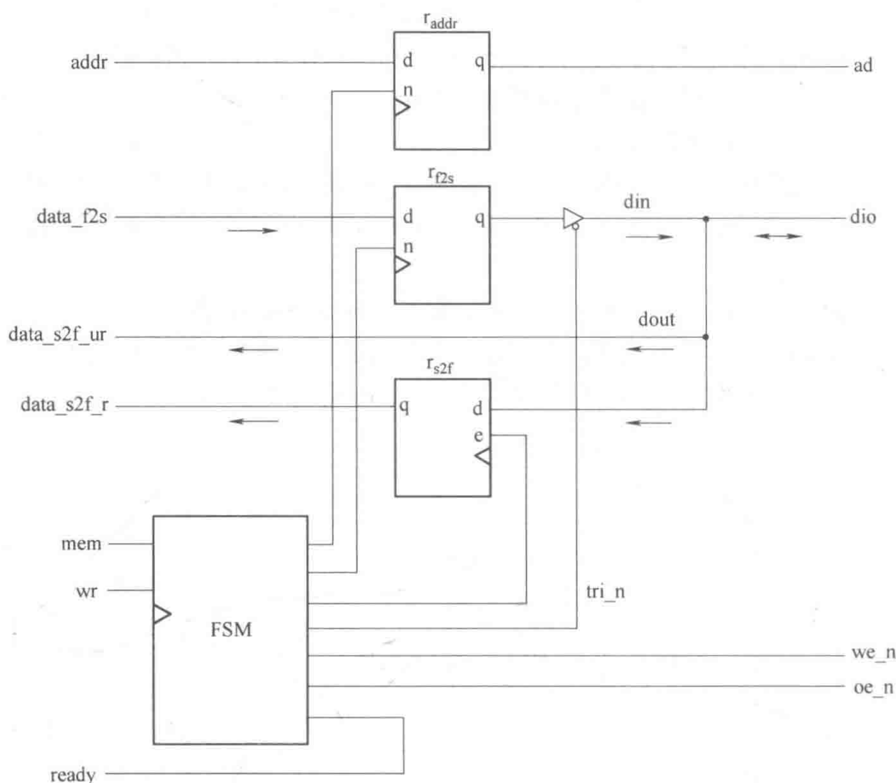


图 11-5　存储控制器的 block 示意图

11.3.2　时序需求

乍一看，时序示意图似乎很复杂，但是实际上这个控制时序是相当的简单。读周期的基本操作顺序如下：

1）将地址信号放到 ad 总线上并使 oe_n 信号有效，以上两个信号在整个操作过程中应保持稳定。

2）最少延迟 t_{AA}，经过这段时间间隔后，SRAM 中的数据将变成为可接收访问的；

3）从 dio 中更新数据，并使信号 oe_n 无效。

在设计中，我们通常使用写使能信号控制写操作周期，如图 11-3a 所示。基本的操作控制顺序如下：

1）将地址信号放到 ad 总线，并将数据信号放在 dio 总线上，激活 we_n 信号。上述信号在整个操作中需保持稳定；

2）等待至少 t_{PWE1}；

3）使 we_n 信号无效，在上升沿上将数据锁存在 SRAM；

4）将数据信号从 dio 总线上移除。

注意到 t_{HD}（在写结束后数据的保持时间）对于本 SRAM 来说为 0ns，这意味着从理论上讲可能将数据移除，同时使 we_n 信号无效。然而，因为传输延时中的变更，这种情形在真实的电路中是无法保证的。为了完成正确的锁存，我们首先需要确保 we_n 信号先于数据无效。

11.3.3　SRAM 的寄存器文件

我们在 4.2.3 节中论述了寄存器文件的设计。它的基本存储单元是 D 触发器，因而它们是完全同步的。尽管 SRAM 外围的存储控制器为同步接口，寄存器文件相对于 SRAM 有以下几点不同之处：

- 寄存器文件通常有一个读端口或者多个读端口；
- 寄存器文件的读端口和写端口可以同时进行访问（即，读操作和写操作可同时进行）；
- 寄存器的写只需要一个时钟周期；
- 从寄存器读端口输出的数据一直是有效的，读操作包括无时钟或者是额外的控制信号。

总的来说，寄存器文件是更加快速、更加灵活的。然而，由于触发器的电路尺寸大小，寄存器文件仅适合于小存储量的存储器。

11.4　安全设计

如图 11-5 中的模块示意图所示，由控制器来产生 SRAM 的读写控制时序。第一个方案选用安全设计，这种设计需准备充裕的时序余量，且不能使用严格的时序约束。控制信号直接由 FSM 产生。控制器用 2 个时钟周期（即：40ns）完成存储器的访问，并且需要 3 个时钟周期（即：60ns）完成相邻交互操作。

11.4.1　ASMD 图

控制器的 ASMD 图可参见图 11-6。图中的状态机有 5 个状态，并以空闲状态为初始状态。当 men 信号有效时，开始存储操作。rw 信号可判定（当前操作）是读操作还是写操作。

对于读操作，状态机跳转到 rd1 状态。在这个跳变的过程中，存储器地址—addr 将会被采集并存储到 addr_reg 存储器中。oe_n 信号在状态 rd1、rd2 中的值是有效的。当读周期结束时，状态机跳回到空闲状态。在跳变的过程中，从 SRAM 中重新取出的数据将会被存入 data_s2f_reg 寄存器，随后 oe_n 信号将会无

效。在图 11-5 所示的 block 示意图中有两个读端口(信号)。data_s2f_r 信号是寄存器输出信号，在状态机退出 r2 状态后其值将变为有效。在下一个读周期结束之前，数据保持不变。data_s2f_ur 信号是直接连接到 SRAM 的 dio 总线数据。它的数据应该在 rd2 状态结束时有效，并且数据会在状态机进入空闲状态之后被移出(总线)。在一些应用中，主系统采样和存取存储器的读出（数据）放入自带的寄存器，若在接近一个时钟周期内也是允许用这种非寄存输出的方式完成。

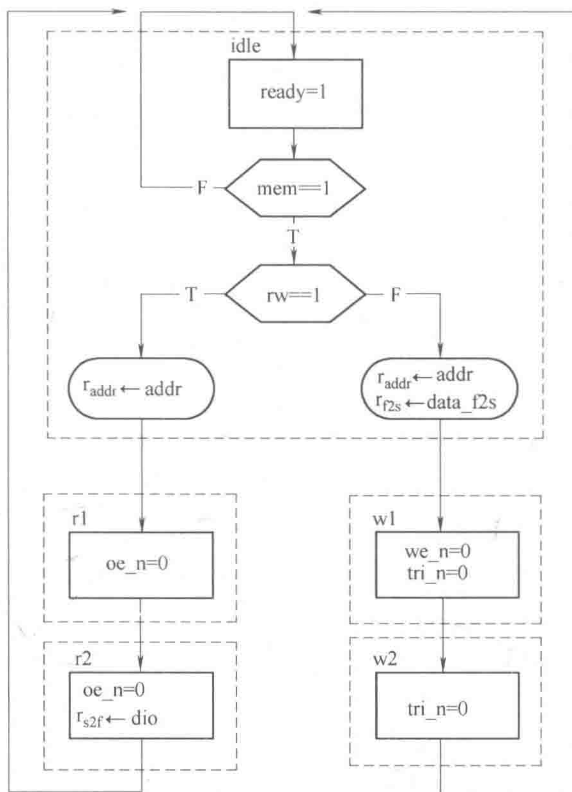


图 11-6　控制器的 ASMD 图

对于写操作，状态机跳转到 wr1 状态。在这个跳变的过程中，存储器地址“addr”和数据“data_f2s”将会分别被采集并存储到addr_reg 和 data_f2s 存储器中。we_n 和 tri_n 信号在 wr1 状态都是有效的。稍后使能三态缓冲器，将数据放到 SRAM 的 dio 总线上。当状态机跳转到 wr2 状态时，we_n 信号无效，但 tri_n 信号依然保持。当 we_n 信号从 0 变为 1 时，这确保了数据被正确地锁存到 SRAM 中。当写周期结束时，状态机跳回到空闲状态，tri_n 信号无效以将数据从 dio 总线上移除。控制器的 ASMD 如图 11-6 所示。

11.4.2　时序分析

为了确保存储控制器的操作正确，我们必须验证该设计适合各种时序需求。重新用 50MHz 的时钟信号控制状态机，因而在每个状态上将保持 20ns。

在读周期期间，对于其中的两个状态，oe_n 是有效的，有效时间共 40ns，即在 10ns 的 t_{AA} 要求上提供 30ns 的时间裕量。尽管这意味着在 rd2 状态 oe_n 可以是无效的，但是这强加了一个更加严格的时序约束。这个问题已在 11.5.3 节描述。当状态机由 rd2 状态跳转到空闲状态时，数据被存储进 data_s2f 寄存器

中。尽管在跳变的时候 oe_n 已经无效，但是因为有 FPGA 的 pad 延时和 SRAM 芯片的 t_{HZOE} 延时，数据仍然会保持一段较短的时间有效。这可以在时钟边沿对数据进行采样。

在写周期期间，we_n 在 wr1 状态无效，则 20ns 时间间隔超出了 8ns 的 t_{PWE1} 的需求。在 wr2 状态，tri_n 信号保持有效，因而可以在 we_n 信号由 0-1 跳变沿期间数据依然保持有效。

在执行期间，读操作和写操作都需要两个时钟周期来完成。在读操作期间，非寄存数据(即，data_s2f_ur)在第二个时钟周期的结尾(即，在第二个时钟周期上升沿之前)有效，寄存数据(即，data_s2f_r)在第二个时钟周期上升沿之后有效。尽管存储器操作可以在 2 个时钟周期内完成，但是主系统不能以这个速度访问存储器。读操作和写操作在操作完成之后都必须返回到空闲状态。主系统必须在等一个时钟周期才能执行新的存储器操作，因此，整个操作过程需要 3 个时钟周期。

11.4.3　HDL 编码(执行)

通过跟随图 11-5 中的 block 示意图和图 11-6 的 ASMD 示意图的思路，可以得到 HDL 代码。存储控制器必须产生快速的、无干扰的控制信号。一种方法是修改输出逻辑单元将摩尔输出信号寄存输出。为了滤除小毛刺和减少时钟到输出的延时时间，这种设计方案为每个输出信号添加一个缓冲器(即，D 触发器)。为了弥补因引入缓冲器导致的一个时钟周期的延时时间，在状态机的输出逻辑单元中，我们使用状态的下一个值(即，state_next 信号)来代替状态的当前值(即，state_reg 信号)。

完整的 HDL 代码如示例 11.1 所示。为了在将来扩充更容易，我们将 S3 版的 2 个 SRAM 芯片分为 A 和 B，并在 SRAM 输入/输出信号的端口声明中加了一个以_a 为后缀的信号。需注意的是，我们使用三态缓冲器来控制双向数据信号 dio_a。

示例 11.1　3 个周期相邻存储器操作的 SRAM 控制器

```
module sram_ctrl
  (
    input wire clk, reset,
    // 主系统的外部输入/输出
    input wire mem, rw,
    input wire [17: 0]addr,
    input wire [15: 0] data_f2s,
    output reg ready,
```

```
  output wire [15: 0] data_s2f_r, data_s2f_ur,
  //SRAM 芯片的外部输入/输出
  output wire [17: 0] ad,
  output wire we_n, oe_n,
  //SRAM 芯片的内部信号
  inout wire [15: 0] dio_a,
  output wire  ce_a_n, ub_a_n, lb_a_n
);
// 状态符号定义
localparam [2: 0]
    idle  =3'b000,
    rd1  =3'b001,
    rd2  =3'b010,
    wr1  =3'b011,
    wr2  =3'b100;
// 信号声明
reg [2: 0] state_reg, state_next;
reg [15: 0] data_f2s_reg, data_f2s_next;
reg [15: 0] data_s2f_reg, data_s2f_next;
reg [17: 0]addr_reg, addr_next;
reg we_buf, oe_buf, tri_buf;
reg we_reg, oe_reg, tri_reg;
// 主体
//FSMD 状态和数据寄存器
always @ (posedge clk, posedge reset)
    if (reset)
      begin
          state_reg  <=idle;
          addr_reg  <=0;
          data_f2s_reg  <=0;
          data_s2f_reg  <=0;
          tri_reg  <=1'b1;
          we_reg  <=1'b1;
          oe_reg  <=1'b1;
      end
```

```
    else
      begin
          state_reg  <= state_next;
          addr_reg  <= addr_next;
          data_f2s_reg  <= data_f2s_next;
          data_s2f_reg  <= data_s2f_next;
          tri_reg  <= tri_buf;
          we_reg  <= we_buf;
          oe_reg  <= oe_buf;
      end
//FSMD 下一状态的逻辑单元
always @ *
begin
    addr_next = addr_reg;
    data_f2s_next = data_f2s_reg;
    data_s2f_next = data_s2f_reg;
    ready = 1'b0;
    case (state_reg)
      idle:
          begin
            if ( ~ mem)
                state_next = idle;
            else
                begin
                  addr_next = addr;
                  if ( ~ rw)// 写
                      begin
                        state_next = wr1;
                        data_f2s_next = data_f2s;
                      end
                  else// 读
                      state_next = rd1;
                end
            ready = 1'b1;
          end
```

```
        wr1:
            state_next = wr2;
        wr2:
            state_next = idle;
        rd1:
            state_next = rd2;
        rd2:
            begin
              data_s2f_next = dio_a;
              state_next = idle;
            end
        default:
            state_next = idle;
    endcase
end
//超前输入逻辑单元
always @ *
begin
    tri_buf = 1'b1;   //下列信号低有效
    we_buf = 1'b1;
    oe_buf = 1'b1;
    case (state_next)
        idle:
            oe_buf = 1'b1;
        wr1:
            begin
              tri_buf = 1'b0;
              we_buf = 1'b0;
            end
        wr2:
            tri_buf = 1'b0;
        rd1:
            oe_buf = 1'b0;
        rd2:
            oe_buf = 1'b0;
```

```
    endcase
  end
  // 输出给主系统
  assign data_s2f_r = data_s2f_reg;
  assign data_s2f_ur = dio_a;
  // 输出给 sram
  assign we_n = we_reg;
  assign oe_n = oe_reg;
  assign ad = addr_reg;
  //sram 芯片的输入/输出
  assign ce_a_n = 1'b0;
  assign ub_a_n = 1'b0;
  assign lb_a_n = 1'b0;
  assign dio_a = ( ~tri_reg) ? data_f2s_reg : 16'bz;
endmodule
```

为了最小化 the off-chip pad delay(在 11.5.1 章有讨论)，应适当地配置相应 FPGA 的 110 个引脚。这可以通过在约束文件中添加附加信息来完成。一种典型的写法是:

NET "ad<17>" LOC = "L3" | IOSTANDARD = LVCMOS33 | SLEW = FAST;

11.4.4　基础测试电路

我们使用两个电路来验证 SRAM 控制器的操作。第一个电路是允许我们手动执行读操作或写操作的基本测试电路。除了 SRAM 芯片的 110 个信号外，这个电路还有下述信号:

- sw：这是一个位宽为 8 位的信号，用来做数据或地址的输入；
- led：这是一个位宽为 8 位的信号，用来显示重新获得的数据；
- btn[0]：当它有效的时候，sw 信号的值为数据寄存器载入的值，对于写操作，寄存器的输出用来做数据的输入；
- btn[1]：当它有效的时候，控制器用 sw 信号的值作为存储器的地址，并执行写操作；
- btn[2]：当它有效的时候，控制器用 sw 信号的值作为存储器的地址，并执行读操作。读出的值输入给 led 信号。

在写操作期间，我们首先指定数据值，并将它加载到内部寄存器，然后指定

地址，开始写操作。在读操作期间，我们指定地址，开始读操作。重新获得的数据在 8 个离散 LED 上显示。完整的 HDL 代码如示例 11.2 所示。

示例 11.2　基础 SRAM 测试电路

```
module ram_ctrl_test
  (
    input wire clk, reset,
    input wire [7: 0] sw,
    input wire [2: 0]btn,
    output wire [7: 0] led,
    output wire [17: 0] ad,
    output wire we_n, oe_n,
    inout wire [15: 0] dio_a,
    output wire ce_a_n, ub_a_n, lb_a_n
  );
  // 信号声明
  wire [17: 0]addr;
  wire [15: 0] data_s2f;
  reg [15: 0] data_f2s;
  reg mem, rw;
  reg [7: 0] data_reg;
  wire [2: 0] db_btn;
  // 主体
  // 实例
  sram_ctrl ctrl_unit
      (.clk(clk), .reset(reset), .mem(mem), .rw(rw),
      .addr(addr), .data_f2s(data_f2s), .ready(),
      .data_s2f_r(data_s2f), .data_s2f_ur(), .ad(ad),
      .we_n(we_n), .oe_n(oe_n), .  dio_a(dio_a),
      .ce_a_n(ce_a_n), .ub_a_n(ub_a_n), .lb_a_n(lb_a_n));
  debounce deb_unit0
      (.clk(clk), .reset(reset), .sw(btn[0]),
        .db_level(), .db_tick(db_btn[0]));
  debounce deb_unit1
      (.clk(clk), .reset(reset), .sw(btn[1]),
```

```
        . db_level( ), . db_tick( db_btn[1]));
debounce deb_unit2
      (. clk(clk), . reset(reset), . sw(btn[2]),
        . db_level( ), . db_tick( db_btn[2]));
// 数据寄存器
always @ (posedge clk)
      if (db_btn[0])
        data_reg <= sw;
// 地址
assignaddr = {10'b0, sw};
//
always @ *
begin
   data_f2s = 0;
   if (db_btn[1])  // 写
         begin
            mem = 1'b1;
            rw = 1'b0;
            data_f2s = {8'b0, data_reg};
         end
   else if (db_btn[2]) // 读
         begin
            mem = 1'b1;
            rw = 1'b1;
         end
   else
         begin
            mem = 1'b0;
            rw = 1'b1;
         end
end
// 输出
assign led = data_s2f[7: 0];
endmodule
```

11.4.5　全面的 SRAM 测试电路

第二个电路提供全面的测试。与其用它验证 SRAM 控制器的操作，不如用它检查 SRAM 芯片的完整性。这个电路有三种功能：

- 以最快的速度将测试样品的数据写入整个 SRAM；
- 以最快的速度读出整个 SRAM 的数据，用原始的样品数据检查重新获得(读出)的数据，记录读出错误数据的数目；
- 写入错误数据。

三种功能可以通过 3 个反跳按钮启动。

ASMD 图如图 11-7 所示。它包含 3 个分支，对应着三种功能。中间的分支将测试样本数据写入 SRAM。wr_clkl、wr_clk2 和 wr_clk3 状态对应着 SRAM 控制器的 idle、wr1 和 wr2 状态。FSMD 使用 18 位的 c 寄存器作为计数器并循环通过该分支 2^{18} 次。在写操作的过程中，c 寄存器中的内容被当做地址，翻转的 16LSB 被当做数据。当循环通过该分支时，FSMD 将数据写入所有存储器的特定区域。左面的分支将数据从 SRAM 控制器中读出。它的 3 个状态对应着 SRAM 控制器的 idle 、rd1 和 rd2 状态。FSMD 再次循环通过分支 2^{18} 次。重新获得的数据和原始测试样品数据比较，并用 err 寄存器记录不匹配数目。右边的分支执行信号的写操作。它用 8 位的开关(switch)形成存储器的地址，向该地址写入错误的样本数据。inj 计数器用来记录注入错误的数目。完整的 HDL 代码如示例 11.3 所示。

示例 11.3　全面的 SRAM 测试电路

```
module sram_test
  (
    input wire clk, reset,
    input wire [7: 0] sw,
    input wire [2: 0]btn,
    output wire [3: 0] an,
    output wire [7: 0] led, sseg,
    output wire [17: 0] ad,
    output wire we_n, oe_n,
    inout wire [15: 0] dio_a,
    output wire ce_a_n, ub_a_n, lb_a_n
  );
  // 状态符号声明
```

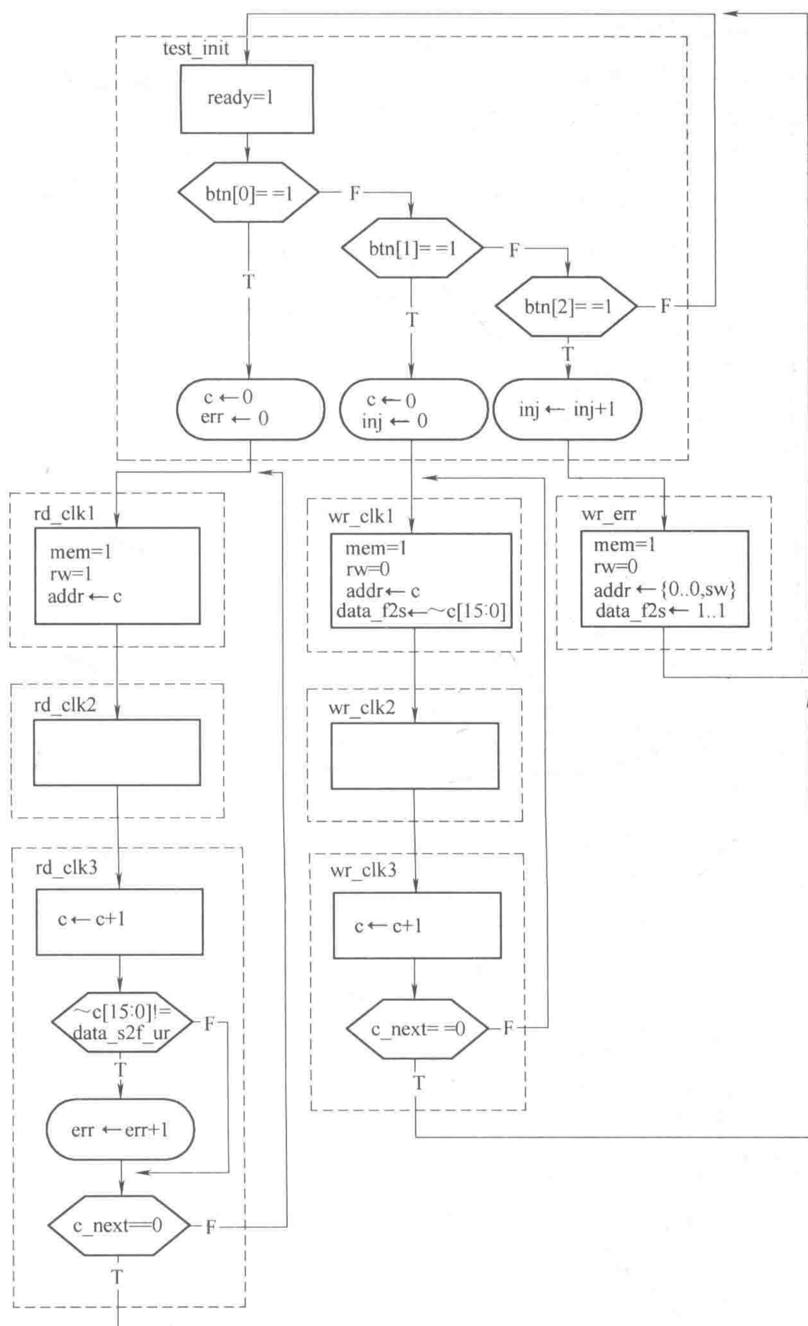


图 11-7　全面的 SRAM 测试电路的 ASMD 图

```
localparam [2: 0]
    test_init = 3'b000,
    rd_clk1   = 3'b001,
    rd_clk2   = 3'b010,
    rd_clk3   = 3'b011,
    wr_err    = 3'b100,
    wr_clk1   = 3'b101,
    wr_clk2   = 3'b110,
    wr_clk3   = 3'b111;
// 信号声明
reg [2: 0] state_reg, state_next;
reg [17: 0]addr;
wire [15: 0] data_s2f;
reg [15: 0] data_f2s;
reg mem, rw;
wire [2: 0] db_btn;
reg [17: 0] c_next, c_reg;
reg [7: 0] inj_next, inj_reg;
reg [15: 0] err_next, err_reg;
// 主体
// ==============================================
// 元件实例
// ==============================================
// 实例
sram_ctrl ctrl_unit
    (.clk(clk), .reset(reset), .mem(mem), .rw(rw),
    .addr(addr), .data_f2s(data_f2s), .ready(),
    .data_s2f_r(), .data_s2f_ur(data_s2f), .ad(ad),
    .we_n(we_n), .oe_n(oe_n), .dio_a(dio_a),
    .ce_a_n(ce_a_n), .ub_a_n(ub_a_n), .lb_a_n(lb_a_n));
debounce deb_unit0
    (.clk(clk), .reset(reset), .sw(btn[0]),
    .db_level(), .db_tick(db_btn[0]));
debounce deb_unit1
    (.clk(clk), .reset(reset), .sw(btn[1]),
```

```
    .db_level(), .db_tick(db_btn[1]));
debounce deb_unit2
    (.clk(clk), .reset(reset), .sw(btn[2]),
    .db_level(), .db_tick(db_btn[2]));
disp_hex_mux disp_unit
    (.clk(clk), .reset(1'b0), .dp_in(4'b1111),
    .hex3(err_reg[15:12]), .hex2(err_reg[11:8]),
    .hex1(err_reg[7:4]), .hex0(err_reg[3:0]),
    .an(an), .sseg(sseg));
//=================================================
//FSMD
//=================================================
//FSMD 状态& 数据寄存器
always @(posedge clk, posedge reset)
    if (reset)
      begin
          state_reg <= test_init;
          c_reg <=0;
          inj_reg <=0;
          err_reg <=0;
      end
    else
      begin
          state_reg <= state_next;
          c_reg <= c_next;
          inj_reg <= inj_next;
          err_reg <= err_next;
      end
//FSMD 下一状态逻辑单元
always @*
begin
    c_next = c_reg;
    inj_next = inj_reg;
    err_next = err_reg;
    addr = 0;
```

```
rw = 1'b1;
mem = 1'b0;
data_f2s = 0;
case (state_reg)
  test_init:
      if (db_btn[0])
        begin
            state_next = rd_clk1;
            c_next = 0;
            err_next = 0;
        end
      else if (db_btn[1])
        begin
            state_next = wr_clk1;
            c_next = 0;
            inj_next = 0;
        end
      else if (db_btn[2])
        begin
            state_next = wr_err;
            inj_next = inj_reg + 1;
        end
      else
        state_next = test_init;
  wr_err: // 写一个error; 在下面2 个时钟中完成
      begin
        state_next = test_init;
        mem = 1'b1;
        rw = 1'b0;
        addr = {10'b0, sw};
        data_f2s = 16'hffff;
      end
  wr_clk1: //sram_ctrl 的待机状态
      begin
        state_next = wr_clk2;
```

```
        mem = 1'b1;
        rw = 1'b0;
        addr = c_reg;
        data_f2s = ~c_reg[15: 0];
    end
wr_clk2: //sram_ctrl 的wr1 状态
    state_next = wr_clk3;
wr_clk3: //sram_ctrl 的wr2 状态
    begin
        c_next = c_reg + 1;
        if (c_next = =0)
            state_next = test_init;
        else
            state_next = wr_clk1;
    end
rd_clk1: //sram_ctrl 的待机状态
    begin
        state_next = rd_clk2;
        mem = 1'b1;
        rw = 1'b1;
        addr = c_reg;
    end
rd_clk2: //sram_ctrl 的rd1 状态
    state_next = rd_clk3;
rd_clk3: //sram_ctrl 的rd2 状态
    begin
        // 比较读出数据; 必须使用非寄存输出
        if ( ~c_reg[15: 0] !  =data_s2f)
            err_next = err_reg + 1;
        c_next = c_reg + 1;
        if (c_next ==0)
            state_next = test_init;
        else
            state_next = rd_clk1;
    end
```

```
      endcase
    end
    // 输出
    assign led = inj_reg;
endmodule
```

注意到写和读不相匹配的数目与七段 LED 显示相连，并以一个 4 位的十六进制数目的形式显示，并且注入错误的数目和 8 个离散的 LED 相连。

我们可以按照如下步骤使用该电路：

- 执行读功能。当 SRAM 未写入数据时，它在初始的"power-on"状态。此时，七段 LED 应显示很多错误。
- 执行写功能。
- 执行读功能。如果 SRAM 控制器和 SRAM 设备工作适当，错误的数目应该是 0。
- 少量的注入错误数据（向不同的存取区域）。
- 再次执行读功能。错误的数目应和注入错误的数目相同。

11.5 更主流的设计

虽然之前的内存控制器功能正常，但性能并不是最佳的。当 SRAM 读周期和写周期都是 10ns 时，该控制器的内存访问占用 60ns（即，3 个时钟周期）。本节中，我们检查几个主流的设计和它们可能存在的时序问题，然后讨论 FPGA 的特点去帮助解决这些问题。

11.5.1 时序问题

异步 SRAM 的时序问题 在高性能异步 SRAM 存储器中有两个敏感的时序问题。第一个问题是 we_n 信号无效。we_n 功能从 0 到 1 变化有点像触发器（FF）的时钟沿，这是数据锁定和存储到内部寄存器单元。注意 SRAM 数据的保持时间（t_{HD}）为 0。虽然它表明在同一时间，这对无效 we_n 和移除数据是允许的，但这种方法在传输延时上的变化是不可靠的。我们必须保证数据从总线移除前 we_n 是无效的。

第二个问题是在数据总线 dio 上可能存在的冲突。数据总线是双向的总线。在写操作期间控制器将数据放置在总线上，在读操作期间 SRAM 将数据放置在总线上。如果控制器和 SRAM 在同一时间将数据放置在总线上，大家知道的一个情况就是读写冲突。为了确保可靠的操作应避免总线冲突。

评估传输延时、设计一个好的存储控制器要求在多信号传输延时上有一个好的理解。然而，这是一个困难的任务。首先，在综合期间，对 RTL 级描述进行优化并映射到互相连接的逻辑单元和线。最终实现的结果可能与初始描述的模块结构图不一致，因而从初始的描述中评估传输延时是困难的。

其次，存储器操作包括芯片外的数据访问。当信号通过 FPGA 的 I/O 脚传输时会引入另外的传输延时。该延时，大家知道有时引脚延时比内部线延时要大很多，且其精确值依赖多种因素，包括 FPGA 器件的类型、输出寄存器的位置(在 LE 或 IOB 输入/输出缓存)、输入/输出标准(I/O)、转换率、驱动强度、外部负载。

这需要很熟悉 FPGA 器件知识和综合软件去完成一个好的时序分析，并评估多种信号的传输延时。

11.5.2　可选设计 I

第一个可选设计的目标是减少总的操作时间。不需要每次操作结束后返回到空闲状态，存储控制器可以在当前存储器操作(即在 rd2 或 wr2 状态)结束时检查 mem 信号并决定下一步做什么。如果存在还没有处理的请求它会立即开始新的存储操作。

图 11-8 中列出了对于该控制器修改后的 ASMD 图。在 rd2 或 wr2 状态，检查 mem 和 rw 信号，如果另一个存储器必须操作 FSMD 有可能立即跳到 rd1 或 wr1 状态。

时序分析　在 11.4.2 节开始介绍的时序分析步骤仍然适用于该设计。然而，当不同类型的随机存储器操作完成时跳过空闲状态会引入敏感的新因素。这个问题就是数据总线上可能存在的冲突。

让我们考虑一下读操作后立即执行写操作。在读操作期间，信号从 SRAM 输入到 FPGA。为了使该操作更容易执行，必须开启 SRAM 的三态缓存(即，数据信号)和关闭 FPGA 的三态缓存(即，高阻)。在写操作期间，信号从 FPGA 输入到 SRAM，三态缓存器的执行方式与读操作中的执行方式正好相反。注意开启或关闭一个三态缓存需要一个小的延时。在 SRAM 芯片中，在图 11-2 中这两个延时定义为 t_{HZOE}(oe_n 到高阻的时间)和 t_{LZOE}(oe_n 到低阻的时间)。

SRAM 控制器开始时，三态缓存在空闲状态。该状态为数据总线提供足够的时间去设置在高阻状态。这两个设计要求两个三态缓存在随机操作器件同时颠倒方向。例如，当从 rd2 状态跳到 wr1 状态时，FSMD 产生信号去关闭 SRAM 的三态缓存和开启 FPGA 的三态缓存。如果 SRAM 的三态缓存关闭太慢或 FPGA 的三态缓存开启太快，在这个转换过程中有可能发生问题。在很小的时间间隔内，双方的缓存可能允许数据放置到总线上，从而发生冲突。同样地，当写操作后立即

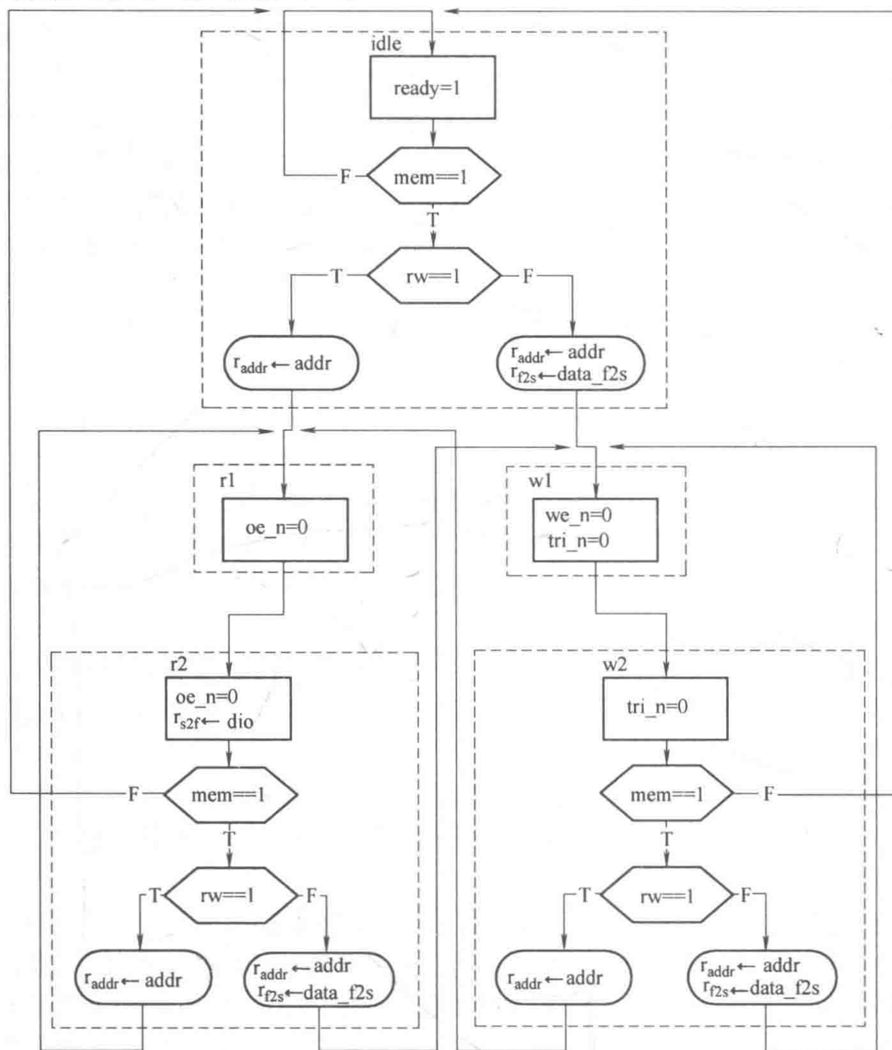


图 11-8　SRAM 控制器设计 I 的 ASMD 图

执行读操作时，也可能发生冲突。

随着读写之间的间隔越来越小，读写之间的冲突对器件没有造成严重的损害，但是有可能引入短暂的大电流，使设计不可靠。我们必须详细地看时序分析报告去检查是否可能发生冲突及微调时序去确定问题。这是个困难的任务，在 11.5.1 节进行讨论。

11.5.3　可选设计 Ⅱ

时序分析在 11.4.2 节进行了描述，它的最初设计提供了较大的安全余量。在该控制器中，存储器操作占用两个时钟周期等于 40ns。因为 SRAM 的读周期和写周期每个是 10ns，我们自然想知道存储器操作周期是否能减少到 20ns。在 ASMD 图中通过除去 rd2 和 wr2 状态就可以做到。第二个可选设计就是使用这个方法。图 11-9 中列出了修改后的 ASMD 图。它占用一个时钟周期完成存储器访问并要求两个时钟周期完成整个操作。

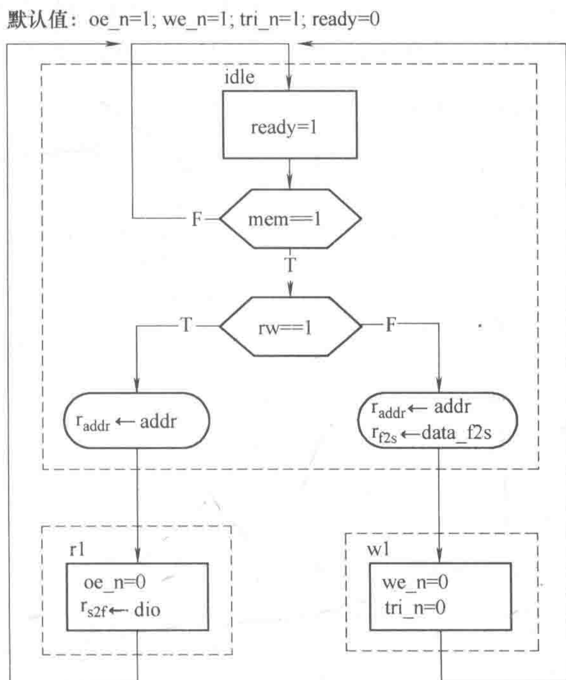


图 11-9　SRAM 控制器设计 Ⅱ 的 ASMD 图

时序分析　从初始控制器中减少一个状态对两个读操作和写操作都增加了更严格的时序约束。让我们首先考虑读操作。该操作期间，地址信号首先通过 FPGA 的 I/O 脚传到 SRAM 的地址总线，并重新得到数据，于是通过 I/O 脚到 FPGA 内部逻辑进行后面的传输。所有操作必须在 20ns 时钟周期内完成。加上 10ns 的 SRAM 地址访问时间，该周期 t_{AA} 必须调整为 2 引脚延时。Spartan-3 器件的引脚延时在从 4ns 到超过 10ns 的范围。因此，我们必须调整综合去完成这个余量。

与读操作不同，写操作是单方向的，只需要传输地址、数据和控制信号到 SRAM 芯片。如果我们假设信号经过类似的引脚延时，延时的绝对值是更小的问题。用键替换信号的正常状态被激活或无效。在 11.5.1 节中进行过讨论，在数据被完全地锁存到 SRAM 之前 we_n 必须无效。在最初的设计中，通过写操作中包含两个状态来完成，wr2，这时 we_n 无效但是数据仍然有效(即 tri_n 仍有效)。在修改的控制器中，we_n 和 tri_n 信号在 wr1 状态结束时同时无效。由于内部逻辑的变化和引脚延时，正常的综合不能够保证在数据从外部数据总线移除前 we_n 无效。此外，对于一个可靠的设计，我们必须调整综合去满足这个目标。

11.5.4　可选设计Ⅲ

我们可以综合两个先前设计中的特性得到第三个可选设计。新的控制器在读操作和写操作中省略第二个时钟周期，并允许随机操作下不返回空闲状态。这是更主流的设计。通过修改先前两个 ASMD 图得到图 11-10。修改后的设计用一个时钟周期完成存储器访问，用另一个时钟周期完成随机操作。

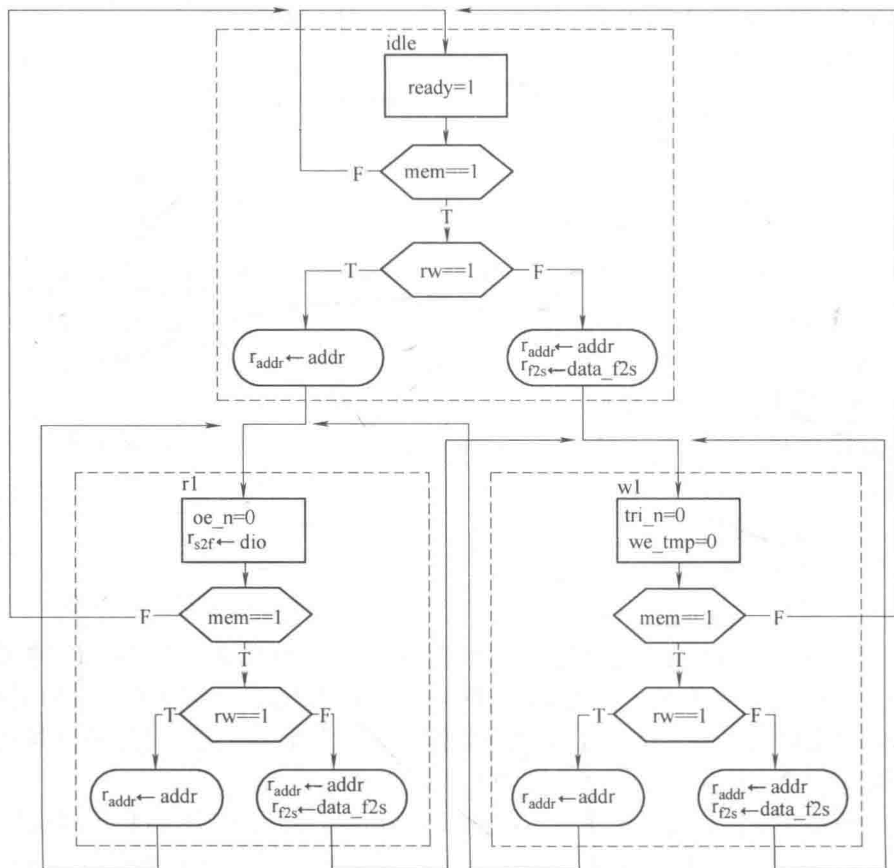


图 11-10　SRAM 控制器设计Ⅲ的 ASMD 图

注意在 ASMD 图中无法显示有效时间不到一个时钟周期的信号 we_n。在 wr1 状态使用 we_tmp，随后的器件中 we_n 源于该信号。

时序分析　因为新设计结合两个先前设计的特点，所有的时钟问题在前面两部分已经讨论，同样的在这个设计中也必须考虑。we_n 信号会产生另外的问题。在随机写操作期间，ASMD 停留在 wr1 状态下。在最初的设计中，we_n 信号是

摩尔状态机的输出。在这个例子中它将会连续地从有效到 0。自从数据被锁定到 SRAM 中 we_n 信号在 0 - 1 之间变化，控制器没有稳定的功能。为了解决这个问题，we_n 信号必须在小部分的时钟周期内有效。

另一个可能解决该问题的方法是在时钟周期的开始一半时使信号有效，10ns 的宽度满足 t_{WPE1} 的需求。直观地，我们试着用选通信号 we_tmp 以及时钟信号 clk 来完成这件事情。

```
assign we_n = we_tmp |  ~clk;
```

然而，由于存在潜在的短时脉冲波形干扰和延时变化，这种做法不是一个可靠的解决方法。更好的可选设计在下一部分进行讨论。

11.5.5　Xilinx 公司的高级 FPGA 特点

本节存储控制器案例阐明了基本 FSM 控制器和同步设计方法的局限性。基本上，FSM 不能产生比时钟信号周期好的控制顺序。这些可选设计的操作依赖某些因素，这是 RTL 级 HDL 描述无法详细说明的。由于传输延时的变化，综合电路是不可靠的，它有可能工作也有可能无法工作。

这有一些 ad hoc 特性去获得更好的控制。这些特性通常依赖于器件本身和软件。例如，数字时钟管理器(DCM)电路和 Spartan-3 器件输入/输出模块(IOB)能够帮助解决一些以前讨论的问题。详细讨论 DCM 和 IOB 已经超过了本书的研究范围。本部分，我们大概了解了一些方法，并举例说明如何去应用这些特性去获得更可靠的控制器。

1. DCM　Spartan-3 的 FPGA 器件包含 8 个数字时钟管理器(DCM)。顾名思义，DCM 是使用系统时钟信号的电路。它能够对引入的时钟信号进行倍频、分频或移相操作产生新的时钟信号。

还有一个得到更好的控制器顺序的方法就是使用快时钟。执行的存储控制器是相当的简单，它本身的电路能够在快时钟速率下操作。例如，我们能够隔离存储控制器并用 DCM 产生 200MHz 时钟信号驱动它，它的周期只有 5ns。图 11-6 中考虑了 ASMD 图的写操作，我们需要扩展 wr1 状态到两个状态并在这些状态中使 we_n 信号有效。现在要求在 4 个状态下完成写操作。然而，由于快时钟速率，4 个时钟周期加起来只有 20ns，这比最初的 60ns 设计要更好。

一个简单的移相时钟应用在下一部分进行讨论。

2. IOB　Spartan-3 FPGA 器件的输入/输出模块(IOB)在 I/O 引脚和器件的内部逻辑之间提供了一个可编程的接口。通过配置类似于驱动电路的存储寄存器和三态缓存来提供不同的转换率、驱动强度及来支持多种 I/O 标准。

11.5.3 节讨论了如何将芯片外引脚延时减少到最小，我们能够强制存储控制器的输出寄存器在 IOB 以及用固定转换率和强度的配置驱动器发送 FF。这可

以在约束文件中通过定义所需的约束和配置做到。

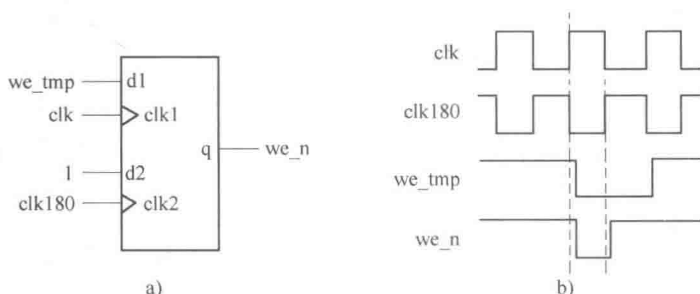


图 11-11　用 DDR 产生半周期信号

IOB 也包括一个双速率(DDR)寄存器，它有两个时钟和两个输入。概念上，我们可以认为它的两个输入是通过两个时钟独立采样而采样值是存储在相同的寄存器中。可以组合 DDR 寄存器和 DCM 去产生一个控制信号，它的宽度是一个时钟信号的小部分，11.5.4 节对 we_n 信号进行了讨论。图 11-11a 为模块图。合格的输出寄存器可以用 DDR 寄存器进行替代。DDR 的顶层部分包含 we_tmp 信号和初始时钟信号 clk。DDR 底部的输出约束为 1 和时钟连接到反相时钟信号 clk180，clk180 是通过 DCM 产生的。在 clk180 信号的上升沿 1 是始终负载，这也是 clk 信号的下降沿。在第二个时钟周期中间 we_n 信号是无效的。时序图见图 11-11b 所示。该方法可以产生一个稳定的半周期信号，比 11.5.4 节门控时钟设计更可靠。

11.6　文献备注

数据手册通过 ISSI 发布，它为 IS61LV25616AL SRAM 器件提供详细的资料。Xilinx 应用注释，XAPP462 在 Spartan-3 FPGA 中使用数字时钟管理器(DCM)，并讨论 DCM 的使用，数据手册，DS099 Spartan-3 FPGA 系列：全部的数据手册，说明 IOB 和 DDR 寄存器的结构和配置。

11.7　实验

11.7.1　512K ×16 配置的存储器

有两个 256K ×16 的 SRAM 芯片，并且它们的 I/O 线路在手动的 S3 板中说明。我们可以把它们扩展为 512K ×16 的 SRAM。

1)结合两个芯片得到一个设计;

2)接着11.4节的程序为512K×16的SRAM设计一个存储控制器，得到HDL描述;

3)为了新控制器修改11.4.5节的实验电路并得到HDL描述;

4)综合实验电路并验证控制器和SRAM芯片的操作。

11.7.2 1M×8配置的存储器

仅仅配置两个像1M×8的SRAM芯片重复11.7.1节中的实验。为了这个目的可以使用1b_n和ub_n信号。

11.7.3 8M×1配置的存储器

256K×16的SRAM单一的bit可以写成如下:

- 读16bit字;
- 在该字中修改指定的bit;
- 回写16bit字。

仅仅配置两个8M×1的SRAM芯片重复11.7.1节中的实验。

11.7.4 扩展存储器实验电路

11.4.5节的存储器实验电路引导不断的随机读操作和随机写操作测试。我们可以扩展该电路使它能够包含不断地在读写操作交替的测试，它能够为整个存储空间测试电路交替写读操作的问题。为了进行更有效的测试，写操作地址和读操作地址应该是不同的。例如，我们可以进行读操作重新得到前16位置写入的数据(如果当前写地址是c，读地址将是c-16)。产生改进的ASMD图，得到HDL描述，综合电路，验证它的操作。

11.7.5 存储控制器和可选设计I的测试电路

在11.5.2节获得可选设计I的HDL代码并产生类似于实验11.7.4的扩展测试电路。综合测试电路并检查操作期间是否有任何错误发生。

11.7.6 存储控制器和可选设计Ⅱ的测试电路

在实验11.7.5中为11.5.3节讨论的可选设计Ⅱ重复操作。

11.7.7 存储控制器和可选设计Ⅲ的测试电路

在实验11.7.5中为11.5.4节讨论的可选设计Ⅲ重复操作。

11.7.8 DCM 的存储控制器

在 DCM 上学习应用记录并随着 11.5.5 节讨论的用高时钟速率(150MHz 或者甚至 200MHz)驱动安全的存储控制器，这在 11.4 节进行了讨论。得到 ASMD 图和 HDL 代码，产生新的实验电路。综合电路并验证存储控制器和 SRAM 的操作。

11.7.9 高性能存储控制器

学习 DCM 和 IOB 文件并应用这些特性去改造 11.5.4 节讨论的可选设计Ⅲ重复操作。产生新的实验电路。综合电路并验证存储控制器和 SRAM 的操作。

第12章　Xilinx Spartan-3 特殊存储器

12.1　简介

数字系统常常需要存储器来进行存储工作。为满足这种需求，大多数 FPGA 设备包含了特定的嵌入式存储器模块。尽管这些模块并不能取代巨大的外部存储器设备，但是它们对于需要中、小型存储器的应用软件很适用。

尽管存储器的基本结构类似，但是它们的 I/O 接口仍然存在很多微小的区别。综合软件想要从代码中获取所需要的特性，或者从潜在的器件库推断出匹配的存储器模块，通常很困难。在 Xilinx ISE 中，我们可以在设计中使用 HDL 实例、Core Generator 程序或者 HDL 行为级模型，在设计中跟嵌入式存储器模型相结合。其中第三种是 semi-device 独立的，在本书中将使用该种办法。在本节中，简要地检查了 Spartan-3 存储器模型、上述的前两种方法，并且提供了关于 HDL 行为级模型的一些关键的细节描述。

12.2　Spartan-3 设备的嵌入式存储器

12.2.1　摘要

在 Spartan-3 设备中有两种嵌入式存储器：分布式 RAM 和块 RAM。分布式 RAM 由逻辑单元的查找表(LUT)构成。查找表(LUT)可以被配置成 16×1 同步 RAM，多个 LUT 可以被级联来形成一个更宽更深的存储器模型。S3 板的 Spartan-3 XC3S200 设备可以提供 30Kbit 的分布式存储，跟块 RAM 或外部存储器相比，该设备的存储量较小。更进一步地说，由于分布式 RAM 使用了逻辑单元，导致这与普通逻辑竞争资源。因此，只有在要用到小型存储器的相关应用时可行。

块 RAM 是一种特殊的嵌入 FPGA 芯片中的存储器模型，从常用的逻辑单元中被分离开。它可以被看作是一种被同步、可配置接口封装的快速 SRAM。每一个块 RAM 包含有 16K(2^{14})数据位加上可选择的 2K 校验位。它可以被定义成不同的位宽，从 16K×1(即 2^{14} 到 2^{0})到 512×32(即 2^{9} 到 2^{5})。Spartan-3 XC3S200 芯片有 12 个块 RAM，共有 172K 数据比特。这些块 RAM 可以被使用在中型应用程

序，比如 FIFO、大型查找表，或中型本地存储器中。作为对比，S3 板的外部 SRAM 芯片有 8Mbit 的容量。

分散式 RAM 和块 RAM 都已经被封装同步接口，因此，不再需要附加存储器控制电路。它们的使用非常灵活，被配置完成单口和双口存储器，并且支持多种类型的缓冲和同步方案。本书不对其进行详细讨论，我们只能验证一些常用的构造，包括同步的单口 RAM、同步的双口 RAM 以及在 12.4 节中提到的 ROM。

12.2.2 对照

S3 板的 Spartan-3 XC3S200 芯片和 S3 板为存储元件提供了一些选项。谨记下列相关选项的作用是很有好处的：

- XC3S200 的寄存器：大约 4.5Kbit，嵌入到逻辑单元和 I/O 缓冲器中；
- XC3S200 的分散式 RAM：30Kbit，由逻辑单元构成；
- XC3S200 的块 RAM：172Kbit，设定为 12 个 16Kbit 模型；
- 外部 SRAM：8Mbit，设定为 2 个 256K × 16 SRAM 芯片。

这些能帮助我们判断出哪个选项最适合常用的应用软件。

12.3 合并存储器模块的方法

尽管存储器模块有着很相似的内部构造，在接口上仍然存在微妙的区别，比如读写端口的数量、同步配置、数据和地址缓冲、使能和复位信号以及初始值。尽管在 HDL 代码中描述期望的行为级模型是有可能的，但是综合软件可能不会识别出设计者的意图。因此，HDL 代码不能总是推断出适当的存储器常常带来很多不便。在 Xilinx ISE 中，设计时有三种方法来合并嵌入式存储器模型：

- HDL 实例；
- Core Generator 程序；
- HDL 行为级模型。

前两种方法对于 Xlinix 设计来说十分详细精确，第三种方法是对嵌入式器件独立的动态描述。由于行为描述清晰，本书中使用该方法。本章节提供了有关于三种方法的简要概述。

12.3.1 元件例化产生的存储器模块

在很多早期的设计实例中，使用 HDL 元件例化来包含初步模型或创建一个层次。除了没有结构体的 HDL 描述，这与例化一个 Xilinx 存储器模块是相似的。我们必须对照指南来找出确切的模型名称和相关参数以及 I/O 端口定义。这是一个单调乏味的步骤，并且对于有着庞大的配置量和设置的存储器来说显然是易于

出错的。

许多 Xilinx 元件实例的代码可以通过选择“Edit -Language Templates”得到。以下是一个 16K ×1 的双口 RAM 的部分代码：

```
//RAMB16-SI-SI: Virtex- Ⅱ / Ⅱ - Pro,
//Spartan-3/3E 16k ×1 Dual-Port RAM
//Xi1inx HDL Language Template version8. 1i
RAMB16-S1-S1# (
. INIT-A (1'b0) ,
. INIT-B(1'b0),
. SRVAL-A (1' b0) ,
. SRVAL-B (1'b0) ,
. WRITE-MODE-A("WRITE-FIRST") ,
. WRITE-MODE-B ("WRITE-FIRST") ,
. SIM-COLLISION-CHECK("ALL"),
. INIT-00 (256' h0…0) ,
. INIT-3F (256' h0…0)
)RAMB16-S1-Sl-inst (
. DOA (DOA) , // 端口A 1bit 数据输出
. DOB (DOB) , // 端口B 1bit 数据输出
. ADDRA (ADDRA) , // 端口A 14bit 地址输入
. ADDRB (ADDRB) , // 端口B 14bit 地址输入
. CLKA (CLKA) , // 端口A 时钟
. CLKB (CLKB) , // 端口B 时钟
. DIA (DIA) , // 端口A 1bit 数据输入
. DIB (DIB) , // 端口B 1bit 数据输入
. ENA (ENA) , // 端口A RAM 使能输入
. ENB (ENB) , / / 端口B RAM 使能输入
. SSRA (SSRA) , // 端口A 同步置位/复位输入
. SSRB (SSRB ) , // 端口A 同步置位/复位输入
. WEA (WEA) , // 端口A 写使能输入
. WEB (WEB) // 端口B 写使能输入
);
```

尽管代码容易获得，但是为了找出正确的元件和合适的配置参数，认真学习指南是必须的。

12.3.2　核生成器产生的存储器模块

为了简化实例化步骤，Xilinx 提供一个有效的程序来生成 Xilinx 特定的元件，被称作核生成器(Coregen)。此功能可以通过选择“Project-New Source”从 ISE 环境中调用。在“New Source”向导对话框弹出后，选择 IP(Coregen & Architecture 向导)调用“Coregen”程序。该程序通过一系列询问并产生一些文件来指引用户。在以 . xco 为扩展名的文本文件中包含了构造存储器元件的必要信息。以 . v 为扩展名的文件包含了用于仿真的“wrapper”代码。这种文件不能被用来例化期望的元件，并且在综合步骤中被忽略。

尽管使用 Coregen 程序比直接 HDL 例化更加方便，但是它不属于 HDL 框架内，并且会导致未完成的设计在 Xilinx ISE 环境中遇到兼容问题。

12.3.3　通过 HDL 生成的存储器模块

尽管建立一个 device-independent HDL 描述是不太可能的，ISE 的综合程序，即 XST，提供了行为级 HDL 模板用于推断 Xilinx FPGA 中的存储模型。这些模板是行为级描述完成的，不包含 device-specific 元件例化描述。他们非常容易被理解，并且可以在没有附加 Xilinx 组成的情况下模拟出来。然而，由于描述没有明确地指出任何 Xilinx 元件，代码有可能不被第三方综合软件认可，不能总是推断出需求的存储器模型。所以，这些模板最好被描述为“semi-portable”和“semi-device-independent”动态描述。常用的存储器模板在 12. 4 节介绍。

另一方面，模板方法的基础是 XST 软件能够识别模板并据此推断出正确的内存模块的能力。软件在升级时改变或曲解某些代码。为了确定我们所需求的模板是否被正确推断，检查 XST 综合报告是很好的途径。

12.4　存储器相关的 HDL 模板

为了使用行为级 HDL 描述来推断出 Xilinx 存储器模型，XST 的模板应该被紧密追踪。为了避免误读，我们应该避免新建属于自己的代码。这些代码在下列的小节中都是建立在 XST V8. 1i 指南的模板上的。除了使用 verilog-2001 进行端口描述或增加地址位宽度和数据位宽度参数，它们与初始的模板是一样的。在独立的 HDL 模块中限制存储器描述是个好的习惯，这样能使模块易于被识别并且在需要时易于被取代。在本小节中，分为 6 个结构讨论行为级 HDL 模板，包括两个单口 RAM，两个双口 RAM，两个 ROM。

12.4.1　单口 RAM

Spartan-3 嵌入式存储器设备已经有了同步接口，这与 11.3 节中描述的一样。它的写操作一直是同步的。在时钟上升沿、地址、输入数据、有关控制信号，写使能被采样。如果写信号有效，执行写操作。(即输入数据在地址信号指定的存储器地址中存储)。

读操作可以是异步的，也可以是同步的。在异步读操作中，地址信号直接用来存取 RAM 队列。地址信号改变后，数据在短暂的延时后可用。在同步读操作中，地址信号在时钟上升沿采样，并被存储在寄存器中。寄存器地址用于访问 RAM 队列。因为寄存器的作用，数据的可用性被延迟，并被时钟信号同步。由于内部构造，异步读操作只可能被分散式 RAM 识别。

异步读功能的单口 RAM　示例 12.1 展示了异步读功能的单口 RAM 的模板。这在 XST 指南的 rams_04 模块之上进行了修改。

示例 12.1　异步读功能的单口 RAM 的模板

```
// 异步读功能的单口RAM
// 在XST 8.1i v_rams_04 基础上有改良
module xilinx_one_port_ram_async
  #(
    parameter ADDR_WIDTH = 8,
              DATA_WIDTH = 1
  )
  (
    input wire clk,
    input wire we,
    input wire [ADDR_WIDTH - 1: 0]addr,
    input wire [DATA_WIDTH - 1: 0] din,
    output wire [DATA_WIDTH - 1: 0]dout
  );
  // 信号声明
  reg [DATA_WIDTH - 1: 0] ram [2 * * ADDR_WIDTH - 1: 0];
  // 实体
  always @ (posedge clk)
      if (we)// 写功能
         ram[addr] <= din;
```

```
  // 读功能
  assign dout = ram[ addr ];
endmodule
```

这段代码与 4. 2. 3 节中讨论的寄存器十分相似，除了在这里读写操作使用同样的地址。它包括一个用来存储的 2 维空间的排列数据类型，并且通过动态指数来存取列表元素。代码显示写操作被时钟信号控制，读操作只由地址决定。由于异步读操作只能被分散式 RAM 识别，这种配置方式只推荐给需求小型存储器的应用软件。

同步读功能的单口 RAM　示例 12. 2 展示了同步读功能的单口 RAM 的模板。这在 XST 指南的 rams_07 模块之上有了改善。

示例 12. 2　同步读功能的单口 RAM 的模板

```
// 同步读功能的单口 RAM
// 在 XST 8. 1i v_rams_07 基础上有改良
module xilinx_one_port_ram_sync
  #(
    parameter ADDR_WIDTH = 12,
              DATA_WIDTH = 8
  )
  (
    input wire clk,
    input wire we,
    input wire [ ADDR_WIDTH - 1: 0 ]addr,
    input wire [ DATA_WIDTH - 1: 0 ] din,
    output wire [ DATA_WIDTH - 1: 0 ]dout
  );
  // 信号声明
  reg [ DATA_WIDTH - 1: 0 ] ram [ 2 * * ADDR_WIDTH - 1: 0 ];
  reg [ ADDR_WIDTH - 1: 0 ]addr_reg;
  // 实体
  always @ ( posedge clk )
  begin
      if ( we )  // 写功能
        ram[ addr ]  <= din;
```

```
    addr_reg  <= addr;
  end
  // 读功能
  assign dout = ram[addr_reg];
endmodule
```

注意到地址信号在时钟上升沿被采样并被存储在 addr_reg 寄存器中，存储器队列(ram 信号)通过 addr_reg 信号被存取。只有在 addr_reg 端口被更新以后数据才可获取，这就意味着其与 clk 信号同步。

综合报告　在综合过程中，一个正确的 RAM 模块应该从代码模块中被生成出来。我们可以对照综合报告来确认 RAM 模块的生成。例如，同步读功能的 4K ×8 RAM 的实例：

```
Xilinx_one_port_ram_sync
    #(.ADDR_WIDTH(12), .DATA_WIDTH(8)) ram_unit_4k_by_8
    (.clk(clk), .we(we), .addr(addr), .din(din), .dout(dout));
```

RAM 的生成应该在综合报告的 HDL 综合部分被指出。

```
==========================================================================
                         * HDL 综合 *
==========================================================================
Found 4096 ×8-bit single-port block RAM for signal <ram>.
```

mode	write-first	
aspect ratio	4096-word × 8-bit	
clock	connected to signal <clk>	rise
write enable	connected to signal <we>	high
address	connected to signal <addr>	
data in	connected to signal <din>	
data out	connected to signal <dout>	
ram_style	Auto	

```
Summary :
inferred 1 RAM(s)
```

块 RAM 的数量通常应该在综合报告的最终报告部分报告：

```
Device utilization summary:
Selected Device : 3s200ft256-5
...
```

```
Number of BRAMs:        2   out of    12     16%
...
```

如我们所期望的，实现了一个 4K×8 单口块 RAM，用两个块 RAM 实现这个电路。

12.4.2 双口 RAM

一个双口 RAM 包括一个用来作为存储通道的第二端口。在理想状态下，第二端口应该能够独立执行读或写操作，并有独立设置其地址、数据输入/输出和控制信号的能力。为了与旧版的 XST 高敏度温感遥感器兼容，我们考虑到一个只能进行读操作的单端口结构。在本书中，双口结构的主要应用就是视频存储，这需要一个写端口和一个读端口。所以，这种结构不会对我们的目标产生严格的约束。由于在单口 RAM 中，一个双口 RAM 的写操作可以是异步的，也可以是同步的。

异步读操作的双口 RAM 示例 12.3 展示了异步读功能的双口 RAM 的模板。这在 XST 指南的 rams_09 模块之上有了改善。

示例 12.3 异步读功能的双口 RAM 的模板

```
// 异步读功能的双口 RAM
// 在 XST 8.1i v_rams_09 基础上有改良
module xilinx_dual_port_ram_async
  #(
    parameter ADDR_WIDTH = 6,
              DATA_WIDTH = 8
  )
  (
    input wire clk,
    input wire we,
    input wire [ADDR_WIDTH - 1: 0]addr_a, addr_b,
    input wire [DATA_WIDTH - 1: 0] din_a,
    output wire [DATA_WIDTH - 1: 0]dout_a, dout_b
  );
  // 信号声明
  reg [DATA_WIDTH - 1: 0] ram [2 * * ADDR_WIDTH - 1: 0];
  // 实体
  always @ (posedge clk)
```

```
    if (we)  // 写功能
      ram[addr_a] <= din_a;
  // 两个读功能
  assign dout_a = ram[addr_a];
  assign dout_b = ram[addr_b];
endmodule
```

写操作与单口 RAM 的相似，但是代码中包含了第二个输出端口 dout_b，该端口从第二个地址 addr_b 中重新获取数据。由于在异步读操作的单口 RAM 中，双口版本只能被分散式 RAM 识别出，因此它的大小是受限制的。如果我们忽略 dout_a 端口，那么这与示例 4.6 中的单端口读寄存器文件相似。

同步读操作的双口 RAM　示例 12.4 展示了同步读功能的双口 RAM 的模板。这在 XST 指南的 rams_11 模块之上有了改善。

示例 12.4　同步读功能的双口 RAM 的模板

```
  // 同步读功能的双口 RAM
  // 在 XST 8.1i v_rams_11 基础上有改良
module xilinx_dual_port_ram_sync
  #(
    parameter ADDR_WIDTH = 6,
              DATA_WIDTH = 8
  )
  (
    input wire clk,
    input wire we,
    input wire [ADDR_WIDTH - 1: 0] addr_a, addr_b,
    input wire [DATA_WIDTH - 1: 0] din_a,
    output wire [DATA_WIDTH - 1: 0] dout_a, dout_b
  );
  // 信号声明
  reg [DATA_WIDTH - 1: 0] ram [2 * * ADDR_WIDTH - 1: 0];
  reg [ADDR_WIDTH - 1: 0] addr_a_reg, addr_b_reg;
  // 实体
  always @ (posedge clk)
  begin
```

```
      if (we)  // 写功能
         ram[addr_a] <= din_a;
         addr_a_reg <= addr_a;
         addr_b_reg <= addr_b;
   end
   // 两个读功能
   assign dout_a = ram[addr_a_reg];
   assign dout_b = ram[addr_b_reg];
endmodule
```

这段代码与示例 12.3 的相似，除了两个地址是首先被存入两个寄存器中，并且寄存器输出端被用来存取存储器。

12.4.3　ROM

尽管 ROM(只读存储器)的取名是一种组合电路，并且没有内部状态。它的输出只取决于输入(例如地址)。在 Spartan-3 设备中没有真正的嵌入式 ROM，但是可以通过一个组合电路或者一个写操作无效的单口 RAM 模拟出。ROM 的内容在 HDL 代码中可以被表示成 case 声明，并在设备运行的时候，它的值被加载到 RAM 中。由于 ROM 建立在 RAM 的基础上，读操作可以是异步的也可以是同步的。

1. 异步读操作的 ROM　由于 ROM 实际上是一种组合电路，因此没有缓冲器和时钟信号。为了与本节中使用的条款协调统一，我们可以称它为异步读操作的 ROM。该类型的 ROM 可以由一个单独的 case 语句陈述，模板如示例 12.5 所示。在代码中简单地重命名了示例 3.14 中的 hex-to-seven-segment LED 解码器输入/输出端口。ROM 函数的地址作为 case 语句的选择表述和相应的内容，被分配到数据信号中。

示例 12.5　异步读操作的 ROM 的模板

```
module rom_template
  (
    input wire [3:0]addr,
    output reg [7:0] data
  );
  // 实体
  always @ *
```

```
  case (addr)
    4'h0: data = 7'b0000001;
    4'h1: data = 7'b1001111;
    4'h2: data = 7'b0010010;
    4'h3: data = 7'b0000110;
    4'h4: data = 7'b1001100;
    4'h5: data = 7'b0100100;
    4'h6: data = 7'b0100000;
    4'h7: data = 7'b0001111;
    4'h8: data = 7'b0000000;
    4'h9: data = 7'b0000100;
    4'ha: data = 7'b0001000;
    4'hb: data = 7'b1100000;
    4'hc: data = 7'b0110001;
    4'hd: data = 7'b1000010;
    4'he: data = 7'b0110000;
    4'hf: data = 7'b0111000;
  endcase
endmodule
```

由于在电路中没有地址和数据缓冲器，该 ROM 不能被块 RAM 识别。这是一种与逻辑单元结合的电路的综合形式，因此，这种类型的 ROM 只能适用于小型表格。

2. 同步读操作的 ROM 对于大型的表格来说，利用一个块 RAM 来实现 ROM 是比较好的方法。由于块 RAM 的读操作被时钟信号控制并与之同步，ROM 同样需要一个时钟信号。该同步读操作的 ROM 模板如示例 12.6 所示。这在 XST 指南的 rams_21c 模板基础上有所改进，我们以 hex-to-seven-segment LED 解码器作为示例。

示例 12.6 同步读操作的 ROM 的模板

```
module xilinx_rom_sync_template
  (
    input wire clk,
    input wire [3:0] addr,
    output reg [7:0] data
```

```
  );
  //信号声明
  reg [3: 0]addr_reg;
  //实体
  always @ (posedge clk)
      addr_reg  <= addr;
  always @ *
      case (addr_reg)
        4'h0: data = 7'b0000001;
        4'h1: data = 7'b1001111;
        4'h2: data = 7'b0010010;
        4'h3: data = 7'b0000110;
        4'h4: data = 7'b1001100;
        4'h5: data = 7'b0100100;
        4'h6: data = 7'b0100000;
        4'h7: data = 7'b0001111;
        4'h8: data = 7'b0000000;
        4'h9: data = 7'b0000100;
        4'ha: data = 7'b0001000;
        4'hb: data = 7'b1100000;
        4'hc: data = 7'b0110001;
        4'hd: data = 7'b1000010;
        4'he: data = 7'b0110000;
        4'hf: data = 7'b0111000;
      endcase
endmodule
```

这段代码与单口 RAM 同步读操作的相似，但是多出一个 case 语句。由于该 ROM 的功能取决于时钟信号，因此它的时序与普通的 ROM 不同。时钟信号的人为定义对于生成 ROM 时用到的块 RAM 是很必要的，在综合过程中，软件自动决定是否使用正规逻辑单元或者块 RAM 来实现电路。

12.5 文献备注

两个 Xilinx 应用程序备注，XAPP464 使用 FPGA 生成的查找表作为 Spartan-3

的分布式 RAM ，XAPP463 使用 FPGA 生成的 Spartan-3 的 RAM，提供了分布式 RAM 和块 RAM 的详细信息。XST 用户手册 v8. li 的的第二章节 HDL Coding Techniques，包括了 24 个 HDL 代码模板来生成多种类型的存储器构造。

ISE In-Depth Tutorial，是比较全面的 ISE 指南，其中一章节介绍了 Core Generator 程序。尽管该程序很简单，为了创建合适的实例，我们还是需要知道模块的基础功能以及相关参数。

12.6 实验

12.6.1 基于块 RAM 的 FIFO

在 4.5.3 节中，我们设计了一个使用寄存器来存储的 FIFO 缓冲器。为了增加它的功能，我们可以用一个块双口 RAM 模块来代替寄存器。为了新的设计生成出 HDL 代码。将 4.5.3 节中讨论的电路与新 FIFO 缓冲器综合，并验证它的功能。注意由于同步读功能而导致新 FIFO 与之前的 FIFO 动作并不完全一致的地方。

12.6.2 基于块 RAM 的栈

在 4.7.7 节讨论了栈的功能，为了增加它的功能，我们可以用一个块双口 RAM 模块来代替寄存器。重复该实验。

12.6.3 基于 ROM 的大量信号地址

我们可以用 $2n \times m$ROM 执行任何 n-输入、m-输出函数。在 3.9.2 节中讨论的大量信号地址，假设 a 和 b 是 4bit 输入信号。电路设计如下：

1）以通用的程序语言写一个程序，如 C 语言或者 Java，组成一个 case 语句，能生成该电路的 28 * 4 真值表；

2）按照示例 12.5 中的 ROM 模板设计该 HDL 代码；

3）综合该电路并验证它的功能；

4）检查综合报告并比较原执行程序和基于 ROM 的执行程序的大小（根据逻辑单元的数量）；

5）将 a、b 扩展为 8bit 信号，重复步骤 1）~4）。

12.6.4 基于 ROM 的 sin(x) 函数

执行 sin(x) 函数的一种方式是使用查找表。假设执行程序需要 10bit 输入协议（即在输入范围 $0 \sim 2\pi$ 内，有 1024 (2^{10}) 个点），以及 8bit 输出协议（即在输入

范围 -1 ~ +1 内，有 256（2^8）点）。使输入信号与输出信号分别为 10bit x 信号和 8bit y 信号。x、y 之间的关系为$\frac{y}{2^7}=\sin\left(2\pi\frac{x}{2^{10}}\right)$。

由于 sin 函数的对称性，我们只需要在第一象限构造一个 $2^8\times7$ 的表$\left(\text{即在 }0\sim\frac{\pi}{2}\text{范围}\right)$，并且使用简单的前后加工电路来获取其他象限的值。电路设计如下：

1)以通用的程序语言写一个程序，组成一个 case 语句，能生成该电路在第一象限的 $2^8\times4$ 查找表；

2)按照示例 12.6 中的 ROM 模板设计该查找表的 HDL 代码；

3)构建一个测试平台来产生 3 个完整周期的 sin 曲线，可以用 10bit 计数器来产生 3×2^{10}时钟循环的 10bit ROM 地址。在 ModelSim 中，可以模拟出 y 信号的形式，以便于仿真出数模转换的影响。

12.6.5　基于 ROM 的 sin(x)和 cos(x)函数

在许多通信调制设计中，sin(x)函数与 cos(x)函数是同时使用的。假设，输入/输出的形式与实验 12.6.4 中一样，新电路有两个输出信号，y_s 与 y_c：

$$\frac{y_s}{2^7}=\sin\left(2\pi\frac{x}{2^{10}}\right)$$

$$\frac{y_c}{2^7}=\cos\left(2\pi\frac{x}{2^{10}}\right)$$

尽管可以按照之前的程序为 cos(x)函数创建一个新的 ROM，但是更好的办法是 sin(x)与 cos(x)函数共享同一个 ROM。这是基于观测得出 cos(x)只是对 sin(x)的函数进行状态的变化，FPGA 的块 RAM 可以提供双口通道。

该电路本质上需要一个双口 ROM，没有这种形式的存储器的 HDL 行为模板。我们需要用 HDL 代码来实验，并检查综合报告，以确定只生成出一个 RAM。为达到该目标，使用 Core Generator 程序或直接将 HDL 元件实例是很有必要的。

构造如上特殊的 ROM，并追踪前后加工电路的 HDL 代码。使用与实验 12.6.4 相似的测试平台来验证电路功能。

第13章　VGA控制器Ⅰ：图形

13.1　简介

VGA（视频图形阵列）是一种在20世纪80年代末由IBM PC引进的视频显示标准，现在已被各种PC图形硬件以及监视器广泛支持。我们在书中讨论的是一种8色640×480分辨率的CRT（阴极射线电子管）基础监视器的设计。本章将验证阴极射线电子管同步以及基础图形处理，第14章将讨论文本生成。

13.1.1　CRT的基本工作方式

黑白CRT监视器概念图如图13-1所示。阴极电子枪发射一束汇聚的电子流，电子流经过真空管而最终打到磷光屏上。发光的同时电子打到屏幕的磷点上。外部视频输入信号的电压强度决定了电子束的密度以及点的亮度，单一信号图如图13-1所示。单一信号为电压值在0～0.7V之间的模拟信号。

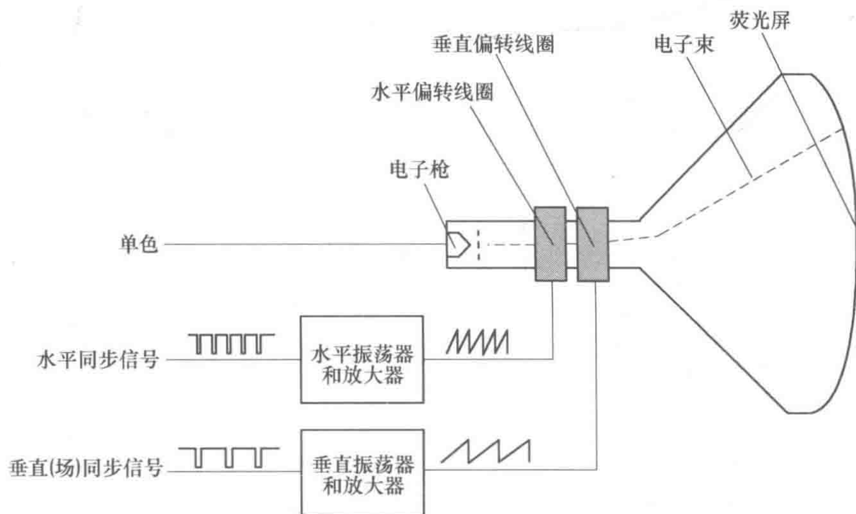


图13-1　CRT监视器运行概念图

真空管外有垂直偏转线圈和水平偏转线圈来产生磁场，控制电子束传播并决定电子束打到屏幕的位置。当今的监视器，电子束在扫描时以系统地固定模式在

屏幕上行进，由左到右并从上到下，如图 13-2 所示。

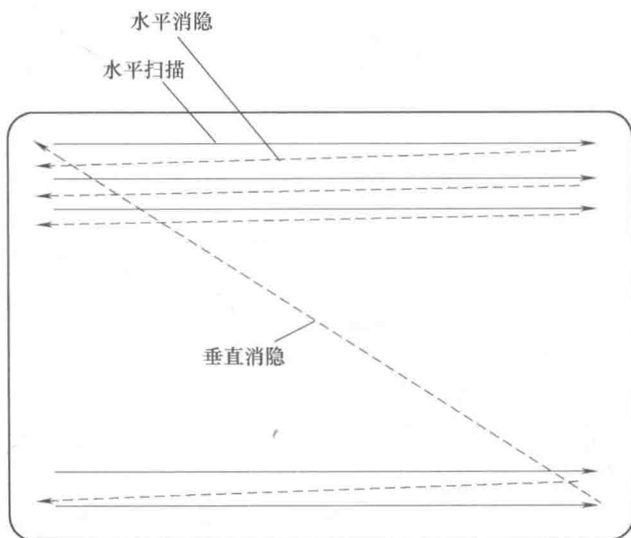


图 13-2　CRT 扫描模式

监视器内部振荡器和放大器产生锯齿波来控制两个偏转线圈。例如，电子束从左边沿移动到右边沿时，水平偏转线圈的电压会逐步增加。电子束移动至右边沿后，当电压值变为 0 时，电子束迅速返回左边沿。锯齿波和扫描的关系如图 13-4 所示。两个外同步信号分别为水平同步(hsync)和垂直同步(vsync)，它们共同控制锯齿波的生成。这些信号为数字信号。水平同步信号和锯齿水平部分的关系同样如图 13-4 所示。注意水平同步信号的“1”和“0”周期与锯齿波的上升下降坡度一致。

彩色 CRT 的基本工作方式与黑白 CRT 工作方式相似，不同的是彩色 CRT 有三束电子束，这些电子束投射红、绿、蓝磷光剂点到屏幕上。这 3 个点组合起来形成一个像素。通过调整三路视频输入信号的电压等级来得到所需的像素点颜色。

13.1.2　S3 板上的 VGA 端口

VGA 端口有五路有效信号，包括水平和垂直同步信号分别为水平同步和垂直同步，以及三路视频信号分别提供红、绿、蓝光。它与一个 15 路引脚超小型 D 连接器物理相连。视频信号为模拟信号，视频控制器使用数字模拟转换器来转换数字输出为所需的模拟量。如果视频信号使用 N 位字表示那么它可以转换为 $2N$ 个模拟量级。三路视频信号可以产生 $23N$ 种不同颜色。同样可知 $3N$ 位决定

了 $3N$ bit 的颜色。在 S3 板内每一路视频信号使用 1bit 字，只能支持 8 种不同颜色。可能的颜色组合见表 13-1。如果我们使用相同的 1bit 信号驱动视频信号，即“000”或“111”，那么监视器的功能就像黑白单色监视器。

表 13-1 3bit VGA 颜色组合

红(R)	绿(G)	蓝(B)	产生的颜色
0	0	0	黑
0	0	1	蓝
0	1	0	绿
0	1	1	青色
1	0	0	红
1	0	1	品红
1	1	0	黄
1	1	1	白

13.1.3 视频控制器

视频控制器生成同步信号以及输出数据的像素序列。VGA 控制器简要模块图如图 13-3 所示。图中包括了一个同步电路(vga_sync)和一个像素生成电路。

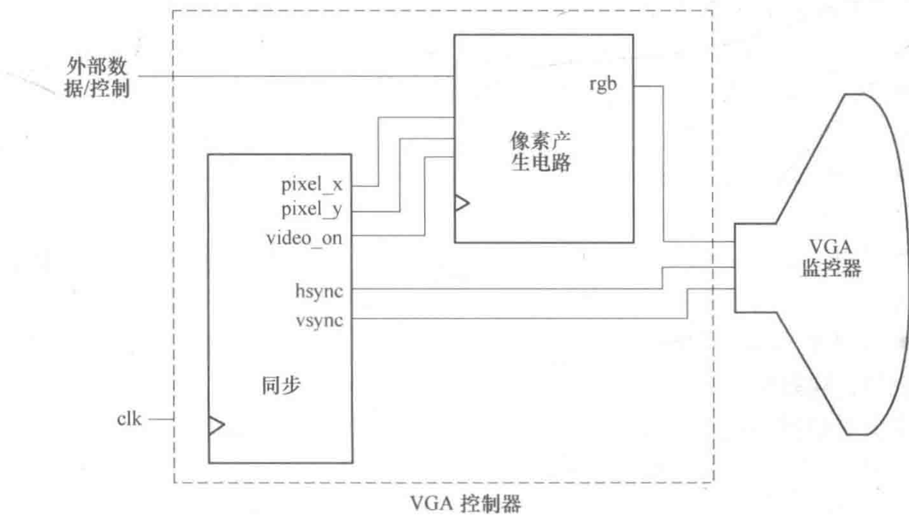


图 13-3 VGA 控制器简要模块图

vga_sync 电路生成时序以及同步信号。水平同步和垂直同步信号连接到 VGA 端口，用来控制监视器的水平、垂直扫描。两路信号在内部计算器中解码，该计算器输出为 pixel_x 和 pixel_y 信号。pixel_x 和 pixel_y 信号标明了扫描相关

位置以及像素点详细的当前位置。vga_sync 电路同时产生 video_on 信号以指出是否使能显示。这个电路的设计在 13.2 节讨论。

像素生成电路生成三路视频信号，这些信号都与三原色信号相关。依据当前像素的坐标(pixel_x 信号，pixel_y 信号)、外部控制及数据信号可以获得一个颜色的值。这个电路的更多相关信息在本章的第二部分以及第 14 章中介绍。

13.2 VGA 同步

视频同步电路生成水平同步信号和垂直同步信号，水平同步信号详细指出了扫过一行所需的时间，垂直同步信号详细指出了扫过整个屏幕需要的时间。随后的讨论都是基于 25MHz 刷新率的 640×480 分辨率的 VGA 屏，25MHz 刷新率的意思是一秒内处理 25M 个像素点。注意该分辨率同样是 VGA 模式。

一个阴极射线管监视器屏幕包括一个小黑边，如图 13-4 的顶部所示。中间的矩形是可见部分。注意垂直轴坐标是向下增加的。左上角以及右下角的坐标分别为(0，0)和(639，479)。

13.2.1 水平同步

一次水平扫描的时序图在图 13-4 中展示。一个水平同步信号周期包含 800 个像素点并且可以分入 4 个区域：

- 显示区域：像素点实际显示在屏上的区域，这个区域的长度为 640 个像素点；
- 折回区域：电子束在此区域返回到左边，视频信号在区域中应为无效即黑色区域，这个区域的长度为 96 个像素点；
- 右边界区域：形成显示区域右边界的区域，也可以认为是前边界即折回前的边界，视频信号在区域中应为无效，这个区域的长度为 16 个像素点；
- 左边界区域：形成显示区域左边界的区域，亦可以认为是后边界即折回后的边界，视频信号在区域中应为无效，这个区域的长度为 48 个像素点。

注意右边界和左边界的长度可能在不同品牌的监视器中有变化。水平同步信号能够用一个专用的模 800 计算器以及一个解码电路得到。这些计数可以在行同步信号的顶端标记，如图 13-4 所示。我们特意从显示区域的起始处开始计数。这样允许我们可以将计数的输出值作为水平(x 轴)的坐标。这个输出组成了 pixel_x 信号。当计数器的输出在 656 和 751 之间时水平同步信号变低。

注意在折回的过程中阴极射线管监视器在左右边界内应该为黑色。我们使用 h_vedio_on 信号来指出当前的水平坐标是否在可显示区域。该信号仅在像素计数小于 640 时为有效。

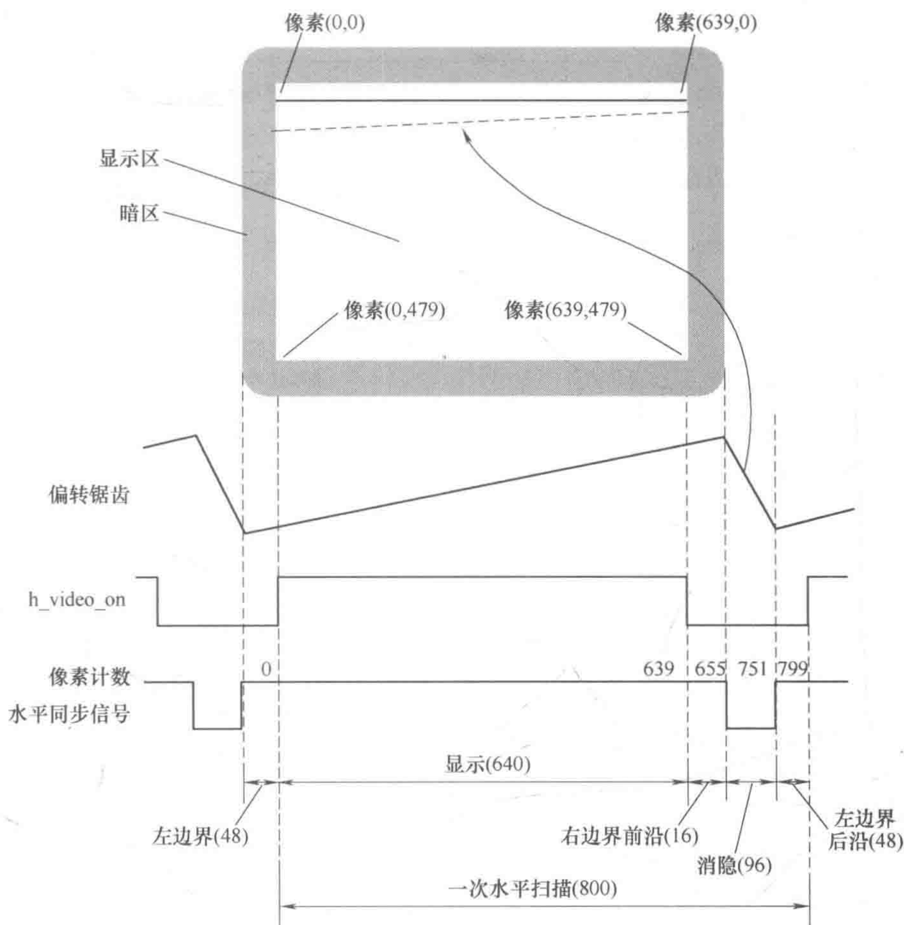


图 13-4　水平扫描时序图

13.2.2　垂直同步

在垂直扫描过程中，电子束逐渐地从顶部运动到底部再返回顶部。这与刷新整个屏幕所需的时间相符。垂直同步信号的形式与水平同步信号的形式相似，如图 13-5 所示。运动的时间单位根据水平扫描行描述。一个垂直同步信号的周期为 525 列并且能够分为 4 个区域：

- 显示区域：水平行实际显示在屏上的区域，这个区域的长度为 480 行；
- 折回区域：电子束返回到屏顶端的区域，视频信号应为无效，这个区域的长度为 2 行；
- 底部边界区域：形成显示区域底部边界的区域，也可以认为是前边界即

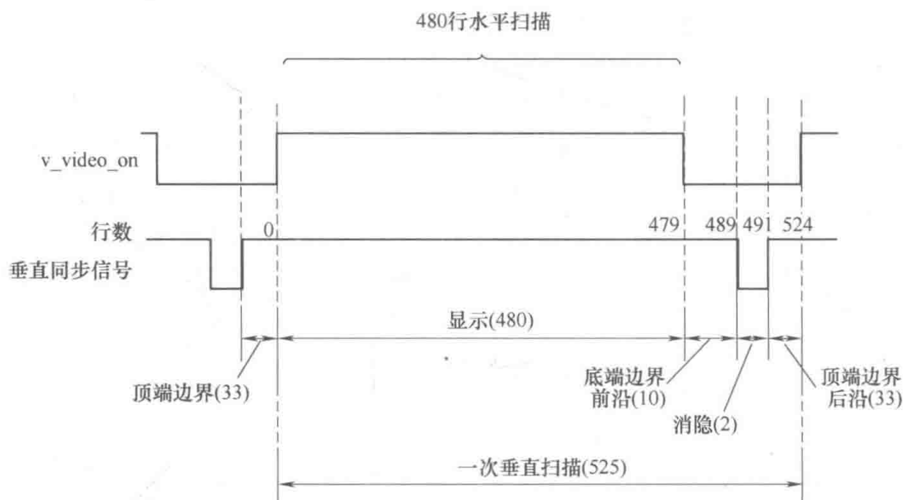


图 13-5　垂直扫描时序图

折回前的边界，视频信号应为无效，这个区域的长度为 10 行；

● 顶部边界区域：形成显示区域顶部边界的区域，也可以认为是后边界即折回后的边界，视频信号应为无效，这个区域的长度为 33 行。

与水平扫描相同，顶端与底端边界的长度由于不同品牌的监视器可能不同。水平同步信号能够通过一个专用模 525 计算器以及一个解码电路得到。同样，我们特意从显示区域的起始处开始计数。这样允许我们可以将计数的输出值作为垂直(y 轴)的坐标。这个输出组成了 pixel_y 信号。当计数器的输出在 490 ~ 491 之间时垂直同步信号变低。

与水平扫描相同，我们使用 v_video_on 信号来指出当前垂直坐标是否在可显示区域。该信号仅在行计数小于 480 时有效。

13.2.3　VGA 同步信号的时序计算

如前文所述，我们假设像素刷新率为 25MHz。刷新率由 3 个参数决定：

p：水平扫描线的像素数。例如 640 * 480 分辨率为

$$p = 800\ \frac{\text{pixels}}{\text{line}}$$

l：一个屏中的行数即垂直扫描的行数。例如 640 ×480 分辨率为

$$l = 525\ \frac{\text{lines}}{\text{screen}}$$

s：每秒的屏数。例如自由闪烁模式，可以表示为

$$s = 60\ \frac{\text{screens}}{\text{second}}$$

参数 s 详细描述了屏刷新的速度。针对于人眼，刷新率必须最少为 30 屏/s 才能让人认为运动是连续的。为了减少闪烁，监视器通常都有一个很高的分辨率，例如如上所述的 60 屏/s。像素率可以由以上 3 个参数算出：

$$\text{pixel rate} = p \times l \times s \approx 25M\ \frac{\text{pixels}}{\text{second}}$$

其他分辨率与刷新率的像素率可以由相似的方式计算出。显然，分辨率与刷新率提升的同时像素率也提升了。

13.2.4　HDL 实现

vga_sync 电路的功能已在 13.1.3 节中讨论。如果系统时钟频率为 25MHz，电路能够被两个专用计数器实现：一个用来保持对水平扫描追踪的模 800 计数器，和一个用来保持对垂直扫描追踪的模 525 计数器。

由于我们的设计通常使用原型板上的 50MHz 晶振，系统时钟频率是像素率的两倍。我们可以生成一个 25MHz 的使能信号以启动或暂停计数来代替违反同步设计方法学的构造一个独立 25MHz 时钟域。这个 tick 信号同样被发送到 p_tick 端口，p_tick 端口作为一个输出信号来调整像素产生电路的操作。

HDL 代码如示例 13.1 所示。它包含一个用于产生 25MHz 的使能单位的模 2 计数器和两个分别用于水平垂直扫描的计数器。我们使用两个状态信号，分别是 h_end 和 v_end，来指示水平和垂直扫描的完成。垂直和水平扫描区域的大小各异，首先将区域大小的值定义为常量。这些常量可以很容易地在当使用了不同分辨率与刷新率时改变。为了消除潜在的故障，水平同步和垂直同步信号插入了输出缓存。这样就造成了一个时钟周期的延时。我们应该在像素产生电路中为 rgb 信号加入一个类似的缓存来补偿这个延时。

示例 13.1　VGA 同步电路

```
module vga_sync
  (
  input wire clk, reset,
  output wire hsync, vsync, video_on, p_tick,
  output wire [9: 0] pixel_x, pixel_y
  );
  // 常量声明
  // VGA 640 ×480 同步参数
```

```
localparam HD = 640; // 水平显示区域
localparam HF = 48; // 水平前(左)边界
localparam HB = 16; // 水平后(右)边界
localparam HR = 96; // 水平折回
localparam VD = 480; // 垂直显示区域
localparam VF = 10; // 垂直前(上)边界
localparam VB = 33; // 垂直后(下)边界
localparam VR = 2; // 垂直折回
// 模2 计数器
reg mod2_reg;
wire mod2_next;
// 同步计数器
reg [9: 0] h_count_reg, h_count_next;
reg [9: 0] v_count_reg, v_count_next;
// 输出缓存
reg v_sync_reg, h_sync_reg;
wire v_sync_next, h_sync_next;
// 状态信号
wire h_end, v_end, pixel_tick;
// 主要结构
// 寄存器
always @ (posedge clk, posedge reset)
    if (reset)
      begin
        mod2_reg <=1'b0;
        v_count_reg <=0;
        h_conut_reg <=0;
        v_sync_reg <=1'b0;
        h_sync_reg <=1'b0;
      end
    else
      begin
        mod2_reg <=mod2_reg;
        v_count_reg <=v_count_next;
        h_count_reg <=h_count_next;
```

```
            v_sync_reg    <= v_sync_next;
            h_sync_reg    <= h_sync_next;
        end
// 使用模2 电路生成25MHz 使能标记
assign mod2_next = ~mod2_reg;
assign pixel_tick = mod2_reg;
// 状态信号
// 水平计数结束(799 )
assign h_end = (h_count_reg == (HD + HF + HB + HR - 1));
// 垂直计数结束(524 )
assign v_end = (v_count_reg == (VD + VF + VB + VR - 1));
// 下一状态水平同步模800 计数器逻辑
always @ *
    if  (pixel_tick)       //25MHz 脉冲
        if (h_end)
          h_count_next = 0;
        else
          h_count_next = h_count_reg + 1;
    else
      h_count_next = h_count_reg;
// 下一状态垂直同步模525 计数器逻辑
always @ *
    if  (pixel_tick & h_end)
        if (v_end)
          v_count_next = 0;
        else
          v_count_next = v_count_reg + 1;
    else
      v_count_next = v_count_reg;
// 水平和垂直同步缓存以避免电磁干扰
//h_sync_next 信号声明在656 ~751 之间
assign h_sync_next = (h_count_reg  > = (HD + HB) &&
                      h_count_reg  <= (HD + HB + HR - 1));
//v_sync_next 信号声明在490 ~491 之间
assign v_sync_next = (h_count_reg  > = (VD + VB) &&
```

```
                    h_count_reg  <= (VD + VB + VR - 1));
 // 视频开或关
 assign video_on = (h_count_reg < HD) && (v_count_reg < VD);
 // 输出
 assign hsync = h_sync_reg;
 assign vsync = v_sync_reg;
 assign pixel_x = h_count_reg;
 assign pixel_y = v_count_reg;
 assign p_tick = pixel_tick;
endmodule
```

13.2.5　测试电路

为了验证同步电路的工作，我们可以将 rgb 信号连接到 3 个开关量上。整个的可视区域应该由一种颜色开启。我们可以仔细检查八种可能的组合来验证表 13-1 中的颜色定义。HDL 代码在示例 13.2 中展示。如 13.2.4 节中提到的，一个输出缓存添加到 rgb 信号中。

示例 13.2　VGA 同步测试电路

```
module vga_test
  (
input wire clk, reset,
input wire [2: 0] sw,
output wire hsync, vsync,
output wire [2: 0]rgb
  );
 // 信号声明
 reg [2: 0]rgb_reg;
 wire video_on;
 // 例化 vga 同步电路
 vga_sync vsync_unit
   (. clk(clk),. reset(reset),. hsync(hsync),. vsync(vsync),
 . video_on(video_on),. p_tick(),. pixel_x(),. pixel_y());
 // rgb 缓存
 always @ (posedge clk, posedge reset)
```

```
  if (reset)
  rgb_reg <=0;
   else
     rgb_reg <= sw;
  // 输出
  assign rgb = (video_on) ? rgb_reg: 3'b0;
endmodule
```

13.3　像素生成电路概述

像素生成电路为 VGA 端口生成了 3bit rgb 信号。外部控制以及数据信号详细说明了屏中的内容，以及 vga_sync 电路输出的 pixel_x 和 pixel_y 信号提供了当前像素的坐标。为了便于我们的谈论，我们将电路分为 3 个宽泛的种类：

- 位图配置；
- 块图配置；
- 对象图配置。

在一个位图配置中，一个视频存储器用来存储将在屏上显示的数据。每个屏内像素直接映射到存储器的一个字中，pixel_x 和 pixel_y 信号来自地址数据。图像处理电路持续不断地更新屏幕并将相关数据写进视频存储器。恢复电路不断地读取视频存储器并且发送数据到 rgb 信号。这种方案已应用于当今高性能的视频控制器中。对于 640×480 分辨率，在一个屏幕上有大约 310 000 即 640×480 个像素点。对于黑白显示，这转化为 310 000 个存储器比特，而对于彩色显示则将转化为 930 000 个存储器比特。一个位图系统例子将在 13.5 节中讨论。

为了减少对存储器的需求，一种可供选择的方法是使用块图配置。在这个方案中，我们收集一组比特组成一个块并将每一块作为一个显示单元。例如，我们可以定义一个 8×8 像素(即 64 个像素点)为一块。640×480 的像素导向屏就变成了 80×60 的块导向屏。这样就只需要 4800 也就是 80×60 个字作为块存储器。每个字的比特数则取决于块类型的数目。例如，如果有 32 种块类型，那么每个字就应该包含 5bit，那么块存储器的大小就是大约 24 000(即 5×4800) bit。块图系统通常需要一个 ROM 来存储块类型。我们称它为类型存储器。假设使用前例提到的单色类型。每一个 8×8 块类型需要 64bit，整个 32 种类型则需要 2000(为 8×8×32) bit。最终整体存储器需求大约为 26 000bit，远小于位图配置的 310 000bit。在第 14 章讨论的文本显示就是基于这个系统。

针对一些应用，视频显示可以非常简单并且只包含一些对象。代替浪费存储

器存储大多数的空白屏，我们可以使用简单的对象生成电路来生成这些对象。我们把这种方法叫做对象图配置。一个对象图案例在 13.4 节中讨论。

这三种配置可以混合在一起来生成一个完整的屏。例如，我们可以使用位图配置来生成背景而后使用对象图配置来产生主要的对象。同样，对一部分屏幕，我们可以使用位图配置来对另一部分屏幕使用块图。

13.4 使用对象映射图的图像生成

对象图像素产生电路包括 3 个对象，其概念图如图 13-6 所示。此图包含 3 个对象产生电路和一个 RGB 信号选择和传送电路。一个对象生成电路完成以下任务：

- 保持当前对象坐标，并且与由信号 pixel_x 与 pixel_y 提供的当前扫描位置进行比较；
- 如果当前扫描位置属于该区域，将信号 obj_i_on 置位以表示当前扫描位置在第 i 个对象的区域上，并且这个对象应该被开启；
- 信号 obj_i_rgb 指定期望的颜色。

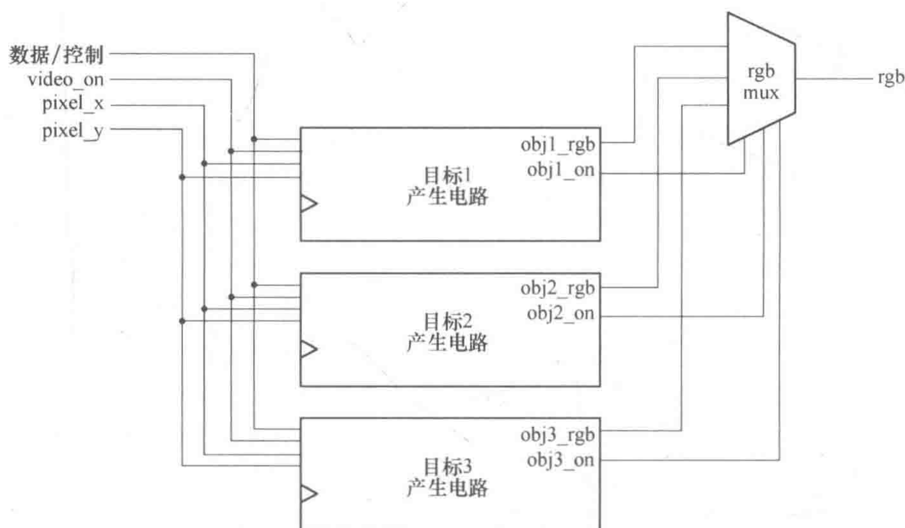


图 13-6　像素生成映射原理图

RGB 选择器依据其内部优选策略进行选择操作。它需要检查多个 obj_i_on 信号并且决定哪个 obj_i_rgb 信号将发送到 RGB 输出端。当多个 obj_i_on 信号同时置位时，优选策略决定了显示命令的先后顺序。它会相应选择一个对象作为前景。

我们使用一个简单化乒乓游戏来举例说明多种图像生成策略。被创建的设计如下：

1）用一个矩形物体创建一个简单静止的屏幕。

2）添加一个圆形物体。

3）引入激励。

4）添加得分和信息文本。

5）创建一个顶层控制电路。

前三个步骤在本节中进行介绍，后两个步骤在第 14 章进行介绍。

13.4.1　矩形对象

一个矩形对象可以在屏幕上通过矩形的边界坐标进行表示。这个游戏的静止屏幕如图 13-7 所示，它包括 3 个对象：墙(图中左侧窄型条纹物)、球拍(图中右侧垂直的短木条)和一个方形球。屏幕中显示区域的坐标如图所示。需注意 y_axis 是向下增长的。

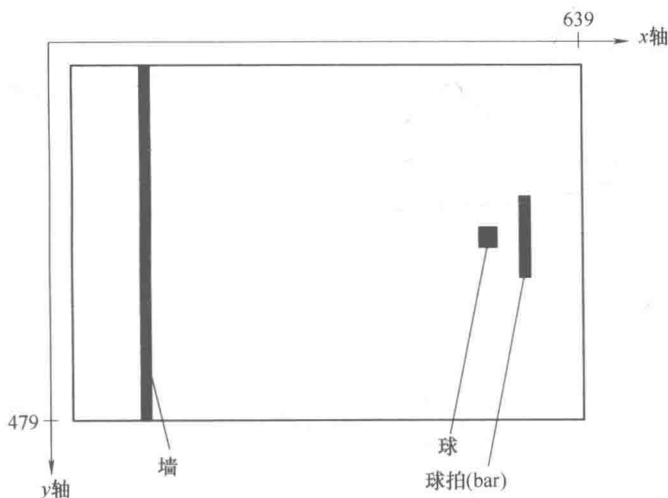


图 13-7　依旧是乒乓游戏画面

首先让我们分析墙条纹的生成。为了表示明确，在代码中，我们定义相关的边界和尺寸为常量。此墙的部分实现代码为

```
// 墙左侧，右侧边界
localparam        WALL_X_L = 32;
localparam        WALL_X_L = 32;
…
```

```
// 墙的像素
assign wall_on =  (WALL_X_L <= pix_x)&&(pix_x  <= WALL_X_R)
// 墙的RGB 输出
assign  wall_rgb = 3'b001; // 蓝色
```

墙是一个 4 个像素宽度的垂直条纹，在 32 列 ~ 35 列之间，被定义为 WALL_X_L 和 WALL_X_R，用于分别表示墙左侧和右侧在 x 轴中的坐标。这个对象有两个输出信号：wall_on 和 wall_rgb。信号 wall_on 表示墙应该开启，在当前水平扫描在这个区域中时，信号 wall_on 被置位。因为此条纹覆盖了整个垂直列，对于 y_axis 的边界是没有需求的。信号 wall_rgb 表示墙的颜色是“001”(蓝色)。

对于木条的部分实现代码如下：

```
// 木条左侧，右侧边界
localparam BAR_X_L = 600;
localparam BAR_X_R = 603;
// 木条顶部，底部边界
localparam  BAR_Y_SIZE = 72;
localparam  BAR_Y_T = MAX_Y/2-BAR_Y_SIZE/2; //204
localparam  BAR_Y_B = BAR_Y_T + BAR_Y_SIZE_ 1;
...
// 木条的像素
assign bar_on = (BAR_X_L <= pix_x) && (pix_x <= BAR_X_R) &&
                (BAR_Y_T <= pix_y)&& (pix_y <= BAR_Y_B);
// 木条 RGB 输出
assign bar_rgb = 3'b010; // 绿色
```

除了包括 y_axis 边界，球拍的代码实现和实现墙代码是一致的。球拍期望的垂直长度为 72 个像素，定义为 BAR_Y_SIZE。因为我们希望把球拍放置在中间，球拍顶部边界被定义为 BAR_Y_T，是 y 轴最大长度的一半减去球拍长度的一半。球拍底部边界是球拍顶部边界加球拍的长度。信号 bar_on 的产生类似于wall_on 信号，但是垂直扫描必须在 y_axis 的边界内。

球的代码可以通过类似的方式进行创建。最后的代码是多路选择器电路，用于检查 3 个对象中开启的信号并发送相应的 RGB 信号到输出端。代码如下：

```
always@ *
    if( ~video_on)
        graph_rgb = 3'b000; // 空白
    else
        if(wall_on)
```

```
            graph_rgb = wall_rgb;
        else if(bar_on)
            graph_rgb = bar_rgb;
        else if(sq_ball_on)
            graph_rgb = ball_rgb;
        else
            graph_rgb = 3'b110; // 黄色背景
```

电路首先检查 video_on 是否被置位，如果是，循环检查3个开启信号。当其中一个开启信号被置位，表示扫描在区域内，并且发送相应的 RGB 信号到输出端。如果没有信号被置位，扫描将处于“背景”模式，并且输出被置于“110”(黄色)。

全部的 HDL 代码实现如示例 13.3 所示。

示例 13.3　对于乒乓游戏画面的像素生成电路

```
module pong_graph_st
  (
    input wire video_on,
    input wire [9:0] pix_x, pix_y,
    output reg [2:0] graph_rgb
  );
  // 常量和信号声明
  //x , y 坐标从(0 , 0 ) 到  (639 , 479 )
  localparam MAX_X = 640;
  localparam MAX_Y = 480;
  //-----------------------------------------
  // 垂直条状物作为墙
  //-----------------------------------------
  // 墙的左侧和右侧边界
  localparam WALL_X_L = 32;
  localparam WALL_X_R = 35;
  //-----------------------------------------
  // 右侧垂直球拍
  //-----------------------------------------
  // 球拍左侧和右侧边界
  localparam BAR_X_L = 600;
```

```
localparam BAR_X_R = 603;
// 球拍顶部和底部边界
localparam BAR_Y_SIZE = 72;
localparam BAR_Y_T = MAX_Y/2-BAR_Y_SIZE/2; //204
localparam BAR_Y_B = BAR_Y_T + BAR_Y_SIZE-1;
//------------------------------------------
// 立方体
//------------------------------------------
localparam BALL_SIZE = 8;
// 立方体左侧和右侧边界
localparam BALL_X_L = 580;
localparam BALL_X_R = BALL_X_L + BALL_SIZE-1;
// 立方体顶部和底部边界
localparam BALL_Y_T = 238;
localparam BALL_Y_B = BALL_Y_T + BALL_SIZE-1;
//------------------------------------------
// 目标输出信号
//------------------------------------------
wire wall_on, bar_on, sq_ball_on;
wire [2:0] wall_rgb, bar_rgb, ball_rgb;
// 实体
//------------------------------------------
// 墙左侧垂直条纹
//------------------------------------------
// 墙的像素
assign wall_on = (WALL_X_L <= pix_x) && (pix_x <= WALL_X_R);
// 墙的RGB 输出
assign wall_rgb = 3'b001; // 蓝色
//------------------------------------------
// 墙右侧垂直条纹
//------------------------------------------
// 球拍的像素
assign bar_on = (BAR_X_L <= pix_x) && (pix_x <= BAR_X_R) &&
                (BAR_Y_T <= pix_y) && (pix_y <= BAR_Y_B);
// 球拍RGB 输出
```

```
  assign bar_rgb =3'b010; // 绿色
  //-------------------------------------------
  // 立方体
  //-------------------------------------------
  // 立方体像素
  assign sq_ball_on =
          (BALL_X_L <=pix_x) && (pix_x <=BALL_X_R) &&
          (BALL_Y_T <=pix_y) && (pix_y <=BALL_Y_B);
  assign ball_rgb =3'b100;    // 红色
  //-------------------------------------------
  //RGB 选择器电路
  //-------------------------------------------
  always @ *
      if ( ~video_on)
        graph_rgb =3'b000; // 空白
      else
        if (wall_on)
            graph_rgb = wall_rgb;
        else if (bar_on)
            graph_rgb = bar_rgb;
        else if (sq_ball_on)
            graph_rgb = ball_rgb;
        else
            graph_rgb =3'b110; // 黄色背景
endmodule
```

从像素产生电路得到相应的信号后，我们将其连接到 VGA 同步电路从而创建完整的视频接口。顶层的 HDL 代码如示例 13.4 所示。注意，信号 graph_rgb 是通过输出缓冲区 buffer 连接到输出端的。当信号 pixel_tick 被置位时，它将被取出。该输出端缓冲器用于与已缓冲的水平同步和垂直同步信号同步。

示例 13.4　依旧是对于一个乒乓游戏画面的完整电路

```
module pong_top_st
  (
    input wire clk, reset,
```

```
   output wire hsync, vsync,
   output wire [2: 0]rgb
  );
  // 信号声明
  wire [9: 0] pixel_x, pixel_y;
  wire video_on, pixel_tick;
  reg [2: 0]rgb_reg;
  wire [2: 0]rgb_next;
  // 实体
  //vga 同步电路例化
  vga_sync vsync_unit
       (.clk(clk),.reset(reset),.hsync(hsync),.vsync(vsync),
       .video_on(video_on),.p_tick(pixel_tick),
       .pixel_x(pixel_x),.pixel_y(pixel_y));
  // 图形生成电路例化
  pong_graph_st pong_grf_unit
       (.video_on(video_on),.pix_x(pixel_x),.pix_y(pixel_y),
       .graph_rgb(rgb_next));
  //RGB 缓冲器
  always @(posedge clk)
       if (pixel_tick)
          rgb_reg  <=rgb_next;
  // 输出
  assign rgb =rgb_reg;
endmodule
```

13.4.2 非矩形对象

直接检测非矩形对象的边界是非常困难的。一个可行的办法是把对象图案使用位图的方式进行详细说明，并且通过位图生成 RGB 信号和开启信号。为了更好地解释，我们举个例子。假设我们想要在兵乓球游戏屏幕中显示一个球，如图 13-8 所示，使用像素为 8×8 圆形位图表示。这个圆形对象生成的步骤如下：

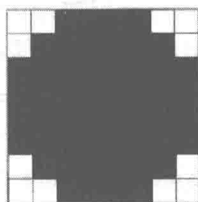


图 13-8 圆形的位图

- 检查扫描坐标是否在 8×8 像素块中；

● 如果是，从位图中获取相应的像素；

● 使用重新获得的像素点为圆形对象产生 RGB 信号和开启信号。

为了执行此方案，我们不得不包含一个图像 ROM 来存储位图和一个地址映射电路用于将扫描坐标转换成 ROM 的行和列。

为了适应此变化，示例 13.3 的部分代码必须修改。首先，我们使用 CASE 语句为这个圆形定义一个图像 ROM，可参考示例 12.5：

```
wire[2 : 0]  rom_addr;
reg[7: 0] rom_data;
...
// 圆形图像
always @ *
case(rom _addr )
    3'h0: rom_data = 8'b00111100; //     * * * *
    3'h1: rom_data = 8'b01111110; //   * * * * * *
    3'h2: rom_data = 8'b11111111; // * * * * * * * *
    3'h3: rom_data = 8'b11111111; // * * * * * * * *
    3'h4: rom_data = 8'b11111111; // * * * * * * * *
    3'h5: rom_data = 8'b11111111; // * * * * * * * *
    3'h6: rom_data = 8'b01111110; //   * * * * * *
    3'h7: rom_data = 8'b00111100; //     * * * *
endcase
```

然后，我们扩展这个球状物生成代码以包括圆形位图的映射。为了方便以后表示活动物体，我们仍旧使用信号来替代球状物边界的常数。其修改后的代码如下：

```
// 圆形像素
assign    sq_ball _on =
          (ball_x_l <= pix_x )&& (pix_x <= ball_x_r)&&
          (ball_y_t <= pix_y)&&(pix_y <= ball_y_b);
// 将当前的像素映射到ROM 中的地址
assign    rom_addr = pix_y[2: 0]-ball_y_t[2: 0];
assign    rom_col = pix_x[2: 0] - ball_x_1 [2: 0];
assign    rom_bit = rom_data [rom_col];
// 球的像素
assign    rd_ball_on = sq_ball_on & rom_bit;
// 球状物RGB 输出
```

```
assign    ball_rgb =3'b100; // 红色
```

第一个语句用于检查当前扫描区域是否在圆形区域内，从而判定是否将信号 sq_ball_on 置位。除了信号用于表示边界外，这段代码和示例 13.3 类似。第二段代码表示通过当前的扫描点获得 ROM 中相应的位。如果扫描点在圆形区域内，需减去顶部边界(ball_y_t)的低三位用于表示 ROM 中相应的行位置(rom_addr)；减去左侧边界(ball_x_l)的低三位用于表示 ROM 中相应的列。通过相应的数组获得最终的位。然后通过和信号 sq_ball_on 进行组合逻辑后产生信号 rd_ball_on。这个设计中仅使用了单色表示圆形区域。我们可以将图像 ROM 复制 3 份用于存储每个像素，并且可产生多种颜色的球体。

最后，我们需要在选择器电路中做一个小的修正，用信号 rd_ball_on 代替信号 sq_ball_on：

```
...
else if(rd_ball_on)
      graph_rgb = ball_rgb;
...
```

在下一节中，这些修改将会组合进活动的画面。

13.4.3 活动的对象

当一个对象在每次扫描中逐渐地改变其位置，它看起来就像是移动的。为了达到此效果，我们使用寄存器来存储对象的边界并且在每次扫描时更新它的值。在游戏中，木条被两个按钮控制，可以移上和移下；球体可以在任何方向上移动和弹跳。在本节中我们将举例说明怎样创建这两个对象的动画。

尽管 VGA 控制器的驱动频率为 25MHz，但是 VGA 监视器屏幕的刷新频率仅仅为 60 次/s。边界寄存器只需要在这个速率下更新。我们创建一个 60Hz 的使能信号 refr_tick，每 1/60s 置位一次。

首先让我们考察木板的设计。为了适应变化的 y 坐标，我们使用信号 bar_y_t 和 bar_y_b 替代常量来描述顶部和底部边界，并且创建一个寄存器 bar_y_reg 来存储当前顶部边界的 y 坐标。当其中一个按钮按下，寄存器 bar_y_reg 在 refr_tick 信号被置位时会以固定值增加或减少。这个固定值定义为常量 BAR_V，且代表木板的速度。我们假设 btn[1] 和 btn[0] 的置位分别表示木板的上升和下降，并且当木板到达屏幕的顶部和底部边界时停止运动。更新寄存器 bar_y_reg 的部分代码如下：

```
// 当前木板y 坐标
always@ *
   begin
```

```
        bar_y_next = bar_y_reg;// 不移动
        if(refr_tick)
            if(btn[1]&(bar_y_b < (MAX_Y_1_BAR_V)))
                bar_y_next = bar_y_reg + BAR_V;// 下移
            elseif(btn[0]&(bar_y_t > BAR_V))
                bar_y_next = bar_y_reg_BAR_V;// 上移
end
```

球状物的设计是相对困难的。我们必须使用 4 个信号代替 4 个边界常量，并且创建 2 个寄存器 ball_x_reg 和 ball_y_reg 用于存储当前左边和顶部边界的 x 和 y 坐标。球状物通常匀速运动（速度和方向都是固定的）。当撞击到墙、木板、屏幕的顶部或底部时，它会改变方向。我们将速度分解为 x 分量和 y 分量，它们的值要么是正数 BALL_V_P，要么是负数 BALL_V_N。两个分量当前的值存储于 x_delta_reg 和 y_delta_reg 寄存器中。更新 ball_x_reg 和 ball_y_reg 寄存器的部分代码如下：

```
// 当前球的位置
assign    ball_x_next = (refr_tick)? ball_x_reg + x_delta_reg:
                          ball_x_reg;
assign    ball_y_next = (refr_tick)? ball_y_reg + y_delta_reg:
                          ball_y_reg;
```

更新 x_delta_reg 和 y_delta_reg 寄存器的代码如下：

```
// 当前球的速度
always @ *
begin
    x_delta_next = x_delta_reg;
    y_delta_next = y_delta_reg;
    if(ball_y_t < 1)// 到达顶部
        y_delta_next = BALL_V_P;
    elseif(ball_y_b > (MAX_Y_1))// 到达底部
        y_delta_next = BALL_V_N;
    elseif(ball_x_1 <= WALL_X_R)// 到达墙
        x_delta_next = BALL_V_P;// 弹回
    elseif((BAR_X_L <= ball_x_r)&&(ball_x_r <= BAR_X_R)&&
        (bar_y_t <= ball_y_b)&&(ball_y_t <= bar_y_b))
        // 到达并撞击右侧木板，球弹回
        x_delta_next = BALL_V_N;
```

```
end
```

注：如果球未触及木板，球将继续向右部区域移动。

完整的代码见示例 13.5。

示例 13.5　对于乒乓游戏动画的像素生成电路

```
module pong_graph_animate
  (
    input wire clk, reset,
    input wire video_on,
    input wire [1: 0]btn,
    input wire [9: 0] pix_x, pix_y,
    output reg [2: 0] graph_rgb
  );
  // 常量和信号声明
  //x, y 的坐标为(0, 0)(639, 479)
  localparam MAX_X =640;
  localparam MAX_Y =480;
  wire refr_tick;
  //------------------------------------------
  // 垂直条纹作为墙
  //------------------------------------------
  // 墙的左、右边界
  localparam WALL_X_L =32;
  localparam WALL_X_R =35;
  //------------------------------------------
  // 右侧垂直木条
  //------------------------------------------
  // 右侧垂直木条左右侧边界
  localparam BAR_X_L =600;
  localparam BAR_X_R =603;
  // 木条顶部和底部边界
  wire [9: 0] bar_y_t, bar_y_b;
  localparam BAR_Y_SIZE =72;
  // 表示顶部边界的寄存器(x坐标固定)
  reg [9: 0] bar_y_reg, bar_y_next;
```

```
// 当开关打开时木板的移动速度
localparam BAR_V = 4;
//---------------------------------------
// 立方体
//---------------------------------------
localparam BALL_SIZE = 8;
// 立方体左右侧边界
wire [9: 0] ball_x_l, ball_x_r;
// 立方体顶部, 底部边界
wire [9: 0] ball_y_t, ball_y_b;
// 表示左侧和顶部的寄存器
reg [9: 0] ball_x_reg, ball_y_reg;
wire [9: 0] ball_x_next, ball_y_next;
// 表示立方体速度的寄存器
reg [9: 0] x_delta_reg, x_delta_next;
reg [9: 0] y_delta_reg, y_delta_next;
// 立方体的速度可为正或负
localparam BALL_V_P = 2;
localparam BALL_V_N = -2;
//---------------------------------------
// 球形
//---------------------------------------
wire [2: 0] rom_addr, rom_col;
reg [7: 0] rom_data;
wire rom_bit;
//---------------------------------------
// 对象输出信号
//---------------------------------------
wire wall_on, bar_on, sq_ball_on, rd_ball_on;
wire [2: 0] wall_rgb, bar_rgb, ball_rgb;
// 实体
//---------------------------------------
// 球形图像ROM
//---------------------------------------
always @ *
```

```
case (rom_addr)
    3'h0: rom_data = 8'b00111100; //   * * * *
    3'h1: rom_data = 8'b01111110; //   * * * * * *
    3'h2: rom_data = 8'b11111111; // * * * * * * * *
    3'h3: rom_data = 8'b11111111; // * * * * * * * *
    3'h4: rom_data = 8'b11111111; // * * * * * * * *
    3'h5: rom_data = 8'b11111111; // * * * * * * * *
    3'h6: rom_data = 8'b01111110; //   * * * * * *
    3'h7: rom_data = 8'b00111100; //   * * * *
endcase
// 寄存器
always @ (posedge clk, posedge reset)
    if (reset)
      begin
          bar_y_reg  <=0;
          ball_x_reg  <=0;
          ball_y_reg  <=0;
          x_delta_reg  <=10'h004;
          y_delta_reg  <=10'h004;
      end
    else
      begin
          bar_y_reg  <=bar_y_next;
          ball_x_reg  <=ball_x_next;
          ball_y_reg  <=ball_y_next;
          x_delta_reg  <=x_delta_next;
          y_delta_reg  <=y_delta_next;
      end
// 当屏幕以60Hz 刷新频率时，在v_sync 的开始同时refr_tick 被置位
assign refr_tick = (pix_y ==481) && (pix_x ==0);
//-----------------------------------------
// (墙)左侧垂直线
//-----------------------------------------
// 墙的像素
assign wall_on = (WALL_X_L <=pix_x) && (pix_x <=WALL_X_R);
```

```
// 墙的RGB 输出
assign wall_rgb = 3'b001; // 蓝色
//-------------------------------------------
// 右侧垂直木条
//-------------------------------------------
// 边界
assign bar_y_t = bar_y_reg;
assign bar_y_b = bar_y_t + BAR_Y_SIZE - 1;
// 木条的像素
assign bar_on = (BAR_X_L <= pix_x) && (pix_x <= BAR_X_R) &&
            (bar_y_t <= pix_y) && (pix_y <= bar_y_b);
// 木条的RGB 输出
assign bar_rgb = 3'b010; // 绿色
// new bar y-position
always @ *
begin
    bar_y_next = bar_y_reg; // 不移动
    if (refr_tick)
      if (btn[1] & (bar_y_b < (MAX_Y-1-BAR_V)))
          bar_y_next = bar_y_reg + BAR_V; // 下移
      else if (btn[0] & (bar_y_t > BAR_V))
          bar_y_next = bar_y_reg - BAR_V; // 上移
end
//-------------------------------------------
// 立方体
//-------------------------------------------
// 边界
assign ball_x_l = ball_x_reg;
assign ball_y_t = ball_y_reg;
assign ball_x_r = ball_x_l + BALL_SIZE - 1;
assign ball_y_b = ball_y_t + BALL_SIZE - 1;
// 球的像素
assign sq_ball_on =
            (ball_x_l <= pix_x) && (pix_x <= ball_x_r) &&
            (ball_y_t <= pix_y) && (pix_y <= ball_y_b);
```

```
// 映射当前像素位置为ROM 中的行列
assign rom_addr = pix_y[2: 0] - ball_y_t[2: 0];
assign rom_col = pix_x[2: 0] - ball_x_l[2: 0];
assign rom_bit = rom_data[rom_col];
// 球的像素
assign rd_ball_on = sq_ball_on & rom_bit;
// 球的RGB 输出
assign ball_rgb = 3'b100;    // 红色
// 当前球的位置
assign ball_x_next = (refr_tick) ? ball_x_reg + x_delta_reg :
                     ball_x_reg;
assign ball_y_next = (refr_tick) ? ball_y_reg + y_delta_reg :
                     ball_y_reg;
// 当前球的速度
always @ *
begin
    x_delta_next = x_delta_reg;
    y_delta_next = y_delta_reg;
    if (ball_y_t < 1) // 到达顶部
      y_delta_next = BALL_V_P;
    else if (ball_y_b > (MAX_Y-1)) // 到达底部
      y_delta_next = BALL_V_N;
    else if (ball_x_l <= WALL_X_R) // 到达墙
      x_delta_next = BALL_V_P;       // 弹回
    else if ((BAR_X_L <= ball_x_r) && (ball_x_r <= BAR_X_R) &&
          (bar_y_t <= ball_y_b) && (ball_y_t <= bar_y_b))
// 到达并撞击右侧木板，球弹回
      x_delta_next = BALL_V_N;
end
//-------------------------------------------
//RGB 选择器电路
//-------------------------------------------
always @ *
    if (~video_on)
      graph_rgb = 3'b000; // 空白
```

```
    else
      if (wall_on)
          graph_rgb = wall_rgb;
      else if (bar_on)
          graph_rgb = bar_rgb;
      else if (rd_ball_on)
          graph_rgb = ball_rgb;
      else
          graph_rgb = 3'b110; // 黄色背景
endmodule
```

和静止屏幕一样，我们可以连接同步电路，并且创建顶层描述。HDL 代码如示例 13.6 所示。

示例 13.6　活动屏幕的完整电路

```
module pong_top_an
  (
    input wire clk, reset,
    input wire [1:0] btn,
    output wire hsync, vsync,
    output wire [2:0] rgb
  );
  // 信号声明
  wire [9:0] pixel_x, pixel_y;
  wire video_on, pixel_tick;
  reg [2:0] rgb_reg;
  wire [2:0] rgb_next;
  // 实体
  // 例化 VGA 同步电路
  vga_sync vsync_unit
      (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
      .video_on(video_on), .p_tick(pixel_tick),
      .pixel_x(pixel_x), .pixel_y(pixel_y));
  // 例化图像产生器
  pong_graph_animate pong_graph_an_unit
```

```
      (.clk(clk), .reset(reset), .btn(btn),
      .video_on(video_on), .pix_x(pixel_x),
      .pix_y(pixel_y), .graph_rgb(rgb_next));
   //RGB 缓冲
   always @(posedge clk)
      if (pixel_tick)
      rgb_reg <= rgb_next;
   // 输出
   assign rgb = rgb_reg;
endmodule
```

注：在此代码中无其他的控制机制，球仅是持续不断地简单移动和弹跳。顶层控制电路将在第 14 章介绍。

13.5　位图映射的图像生成

位映射将每个像素点映射成视频存储器中的一个字节。在分辨率为 640 × 480 的屏幕中约含有 310K 个像素。可以分别转换成 310K 用于单色显示或是 930K 用于彩色显示。视频存储器的实际容量应该更大，因为存储地址必须适当组合以便于快速存取。例如，将像素的当前坐标映射到存储器中的位置，我们可以连接位宽为 10 位的 x 坐标和位宽为 9 位的 y 坐标。这个步骤不需要额外的电路将像素的坐标转换为存储地址，而是传入存储器中没有使用的空间。存储器的容量大小应由 310K 字节上升至 512K 字节。

对于 S3 电路板，如第 11 章和第 12 章介绍的，存储器可以使用外部 SRAM 芯片或者 FPGA 内嵌的块 RAM。考虑到 Spartan 3S200 系列芯片中的 RAM 块的容量仅有 192kbit/s。对于整个屏幕显示的位图容量是不够的。为了达到目的，我们必须使用外部的 8M bits 的 SRAM。

在本节中，我们使用一个小的 128 × 128 的屏幕区域来举例说明位图方案的设计。这个屏幕有 16K(2^{14})个像素，并且需要一个 16 × 3 的视频存储器用于色彩显示。这将通过 3 个内嵌 RAM 块来实现。这个小区域是屏幕左上角的部分，用于显示一个活动的像素点的轨迹，如图 13-9 所示。这个电路使用了一个 3 位的用于指定轨迹颜色的开关，和一个用于随机选择轨迹起点的按钮开关。当按钮开关被按下时，这个点开始移动，就像 13.4.3 节中活动的球。在这个点撞击这个区域中的 4 个边后，轨迹形成矩形。在按钮开关被按下的每个时刻，将产生新的轨迹。

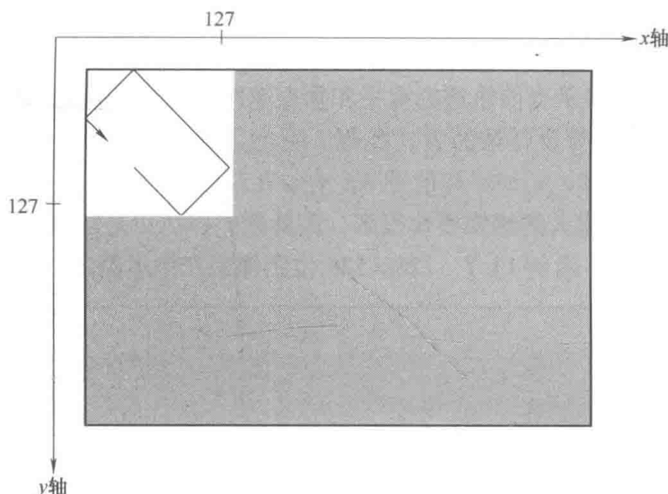


图13-9　在128×128的位图中的轨迹点

13.5.1　双口RAM实现

双口RAM实现的原理图如图13-10所示，视频存储器是一个同步的16K×3 (2^{14}×3)的双口RAM。双口模块(详见示例12.4)可以用于实现双口RAM。存储器的高7位地址是像素y坐标的低7位，存储器的低7位地址是像素x坐标的低7位。dot_xy电路用于跟踪圆点的轨迹，并且产生当前的坐标x和坐标y，并将两者联合作为写地址。信号sw为外部开关输入，位宽为3位，用于表示RGB的值，和存储器的din_a端口相连，像素y的低7位和像素x的低7位构成了读地址。数据不断重新获取并且将相应的数据发送给RGB选择器电路。

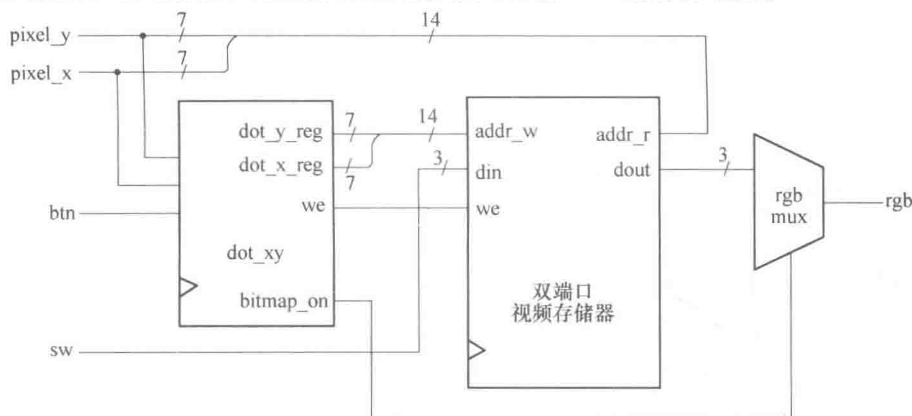


图13-10　轨迹点电路原理图

圆点轨迹像素产生电路的完整代码如示例 13.7 所示。我们使用两个寄存器 dot_x_reg 和 dot_y_reg 来表示圆点轨迹当前的坐标 x 和坐标 y，使用两个寄存器 v_x_reg和 v_y_reg 表示当前轨迹的水平和垂直速度。圆点坐标值和速度的计算方法和 13.4.3 节中计算反弹球的方式类似。除此之外，数据会定时更新，在按钮开关开启时，信号 dot_x_next 和信号 dot_y_next 获取 pix_x 和 pix_y 的低 7 位。由于这些信号的变化比人的感知要快很多，因此新坐标可以是随机值。

示例 13.7　128×128 位图像素产生电路

```
module bitmap_gen
  (
    input wire clk, reset,
    input wire video_on,
    input [1: 0]btn,
    input [2: 0] sw,
    input wire [9: 0] pix_x, pix_y,
    output reg [2: 0] bit_rgb
  );
  // 常量和信号声明
  wire refr_tick, load_tick;
  //-------------------------------------------
  // 视频SRAM
  //-------------------------------------------
  wire we;
  wire [13: 0]addr_r, addr_w;
  wire [2: 0] din, dout;
  //-------------------------------------------
  // 圆点位置和速度
  //-------------------------------------------
  localparam MAX_X = 128;
  localparam MAX_Y = 128;
  // 圆点速度可为正也可为负
  localparam DOT_V_P = 1;
  localparam DOT_V_N = -1;
  // 圆点轨迹位置寄存器
  reg [6: 0] dot_x_reg, dot_y_reg;
```

```
wire [6: 0] dot_x_next, dot_y_next;
// 圆点速度轨迹寄存器
reg [6: 0] v_x_reg, v_y_reg;
wire [6: 0] v_x_next, v_y_next;
//------------------------------------------
// 输出信号
//------------------------------------------
wire bitmap_on;
wire [2: 0] bitmap_rgb;
// 实体
// 为按钮例化弹起电路
debounce deb_unit
   (. clk(clk), . reset(reset), . sw(btn[0]),
     . db_level(), . db_tick(load_tick));
// 例化双口 RAM
xilinx_dual_port_ram_sync
     #(. ADDR_WIDTH(14),. DATA_WIDTH(3)) video_ram
     (. clk(clk),. we(we),. addr_a(addr_w),. addr_b(addr_r),
     . din_a(din),. dout_a(),. dout_b(dout));
// 视频RAM 接口
assign addr_w = {dot_y_reg,dot_x_reg};
assign addr_r = {pix_y[6:0],pix_x[6:0]};
assign we = 1'b1;
assign din = sw;
assign bitmap_rgb = dout;
// 寄存器
always @ (posedge clk,posedge reset)
     if (reset)
        begin
             dot_x_reg  <=0;
             dot_y_reg  <=0;
             v_x_reg  <= DOT_V_P;
             v_y_reg  <= DOT_V_P;
        end
     else
```

```
      begin
          dot_x_reg  <= dot_x_next;
          dot_y_reg  <= dot_y_next;
          v_x_reg  <= v_x_next;
          v_y_reg  <= v_y_next;
      end
// 在v-sync 开始的一个tick 被置位
assign refr_tick = (pix_y ==481) && (pix_x ==0);
// 位图区域像素
assign bitmap_on = (pix_x <= 127) & (pix_y <= 127);
// 圆点位置
// 当btn [0 ]开启时，随机下载圆点位置
assign dot_x_next = (load_tick) ? pix_x[6: 0] :
                    (refr_tick) ? dot_x_reg + v_x_reg :
                    dot_x_reg;
assign dot_y_next = (load_tick) ? pix_y[6: 0] :
                    (refr_tick) ? dot_y_reg + v_y_reg :
                    dot_y_reg;
// 圆点 x坐标速度
assign v_x_next =
      (dot_x_reg ==1) ? DOT_V_P :            // 到达左侧
      (dot_x_reg == (MAX_X-2)) ? DOT_V_N : // 到达右侧
      v_x_reg;
// 圆点 y坐标速度
assign v_y_next =
      (dot_y_reg ==1) ? DOT_V_P :            // 到达顶部
      (dot_y_reg == (MAX_Y-2)) ? DOT_V_N : // 到达底部
      v_y_reg;
//-------------------------------------------
//RGB 选择器电路
//-------------------------------------------
always @ *
    if ( ~video_on)
      bit_rgb =3'b000; // 空白
    else
```

```
        if (bitmap_on)
            bit_rgb = bitmap_rgb;
        else
            bit_rgb = 3'b110; // 黄色背景
endmodule
```

系统顶层的 HDL 代码如示例 13.8 所示。

示例 13.8　位图屏幕完整电路

```
module dot_top
  (
    input wire clk, reset,
    input wire [1: 0]btn,
    input wire [2: 0] sw,
    output wire hsync, vsync,
    output wire [2: 0]rgb
  );
  // 信号声明
  wire [9: 0] pixel_x, pixel_y;
  wire video_on, pixel_tick;
  reg [2: 0]rgb_reg;
  wire [2: 0]rgb_next;
  // 实体
  // 例化 VGA 同步电路
  vga_sync vsync_unit
      (.clk(clk),.reset(reset),.hsync(hsync),.vsync(vsync),
      .video_on(video_on),.p_tick(pixel_tick),
      .pixel_x(pixel_x),.pixel_y(pixel_y));
  // 例化图像发生器
  bitmap_gen bitmap_unit
      (.clk(clk),.reset(reset),.btn(btn),.sw(sw),
      .video_on(video_on),.pix_x(pixel_x),
      .pix_y(pixel_y),.bit_rgb(rgb_next));
  //RGB 缓冲器
  always @ (posedge clk)
```

```
      if (pixel_tick)
         rgb_reg  <= rgb_next;
   // 输出
   assign rgb = rgb_reg;
endmodule
```

13.5.2 单口 RAM 实现

尽管双口存储器是理想的，但却不总是可用的。使用常规的单口存储器(例如 S3 板上的外部 SRAM)用于视频存储需要注意在数据获取时读写操作间的冲突。为了说明此问题，我们将内嵌的 RAM 块配置成一个单口同步 SRAM 并且重新设计之前的圆点轨迹电路。

在圆点轨迹电路中，圆点坐标在每次屏幕扫描时更新一次。因此可以以这样的速度对视频存储器进行写操作。我们可以在因视频关闭而垂直线折回的这个期间完成这些操作，并且不会发生视频存储器写操作与屏幕数据读取操作的冲突。注意当 pixel_y 为 481 时，信号 refr_tick 将被置位。在此位置时视频被关掉，并且写视频存储器不会干扰屏幕数据的重获取。我们使用 we 信号作为单口 RAM 的写使能。已经在示例 12.2 讨论过的单口 RAM 模块在这里可以使用。示例 13.7 中的存储部分代码现在变成了：

```
   // 例化双口RAM
   xilinx-one-port-ram-sync
   #(.ADDR-WIDTH(14),.DATA-WIDTH(3))video-ram
   (.clk(clk),.we(we),.addr(addr),
   .din(din),.dout(dout));
   // 视频RAM 接口
   assign addr_w = (dot_y_reg,dot_x_reg);
   assign addr_r = (pix_y C6:01,pix_x[6:01);
   assign addr = (refr_tick)? addr_w:addr_r;
   assign we = refr_tick;
   assign din = sw;
   assign bitmap_rgb = dout;
```

圆点轨迹电路在每次屏幕扫描时更新一个像素点。因为写操作需要的存储器带宽为 60 × 3bit/s，所以这种电路的更新速度非常慢。因此，之前的设计过于简单。当需要的存储器带宽比较大时(例如当屏幕的很大一部分在很快的刷新率下更新)，存储接口的设计就变得相当困难。

13.6 文献备注

詹姆斯编写的《数字系统原型设计速成》中描述了关于时序信息监控。

13.7 实验

13.7.1 VGA测试图案发生器

一个VGA测试图案发生器产生两个简单的图案用于验证VGA监视器的操作是否正确。第一个图案将屏幕平均分成8个垂直的条纹，每个条纹显示唯一的颜色。第二个图案与之类似但是将屏幕分成8个水平条纹。一个1位选择开关选择不同图案。

为图案发生器设计一个像素生成电路，并且和一个同步电路在顶层相连。综合并验证电路的操作是否正确。

13.7.2 SVGA模式同步电路

72Hz刷新速率的超级VGA模式的说明如下：

- 分辨率：800×600像素；
- 像素频率：50MHz；
- 水平显示区域：800像素；
- 水平右侧边界：64像素；
- 水平左侧边界：56像素；
- 水平折回：120像素；
- 垂直显示区域：600线；
- 垂直右侧边界：64线；
- 垂直左侧边界：56线；
- 垂直折回：120线。

我们希望创建一个双模式同步电路，既能支持VGA模式也能支持SVGA模式。模式选择通过开关实现。构建电路的步骤如下：

1）修改示例13.1中的水平和垂直同步计数器以适应两种模式。

2）设计一个像素生成电路能够在屏幕上画一个100像素网格(即，每100像素分别画一条垂直和水平线)。

3）生成顶层模块。综合并验证两个模式的操作的正确性。

13.7.3 可视化屏幕调整电路

由于监视器内部时序错误，屏幕的显示部分并不总是居中的。我们可以通过微调黑色区域边界的宽度来调节显示部分的位置。在水平扫描线中，有 64 个像素表示右侧和左侧边界。为了水平地移动显示部分，我们可以在边界一侧增加一定数量的像素，并且在相反的一侧减去相同数量的像素。使用同样的方法我们可以调节显示部分进行垂直移动。设计一个屏幕调节电路的步骤如下：

1）扩展 VGA 同步电路包括如下特征：使用开关选择垂直或水平模式，并且使用两个按钮来控制显示屏幕的左/上和右/下移动；

2）修改 13. 2. 5 节中的测试电路以包含新的同步电路；

3）综合并验证电路操作的正确性。

13.7.4 箱子里球的电路

“箱子里球”的电路显示了在立方体箱子里的一个活动球。这个立方体箱子在屏幕内居中，其尺寸为 256 ×256 像素。这个球是一个 8 ×8 像素大小的圆球。当这个球撞击到墙壁，这个球反弹并且墙闪光(比如，暂时改变颜色)。这个球可以以 4 种不同速度运动，并且可以通过两个滑动开关选择。当一个按钮开关开启时，球的方向的改变是随机的。产生源代码 HDL，然后综合并验证电路操作的正确性。

13.7.5 箱子里两个球的电路

我们将 13. 7. 4 节试验中的电路扩展为包括箱子里包含两个球。当两个球撞击，两个球依据物理定律将会产生新的方向。生成源代码 HDL，然后综合并验证电路操作的正确性。

13.7.6 两个游戏者的游戏

两个游戏者的游戏用另外的短板替代了左侧的墙，并且短板由第二名游戏者控制。为了能提供两个游戏者，我们可以使用 9. 4 节中的键盘接口作为输入。4 个按键可以控制两个短板的垂直运动。生成源代码 HDL，然后综合并验证电路操作的正确性。

13.7.7 越狱游戏

越狱游戏类似于乒乓游戏。在此游戏中，左侧的墙被替代为多层的砖。当球撞击到砖时，球被弹回并且消失。基本的屏幕如图 13-11 所示。代码如示例 13. 5 所示，我们假设游戏在不断地运行中。产生源代码 HDL，然后综合并验证电路

操作的正确性。

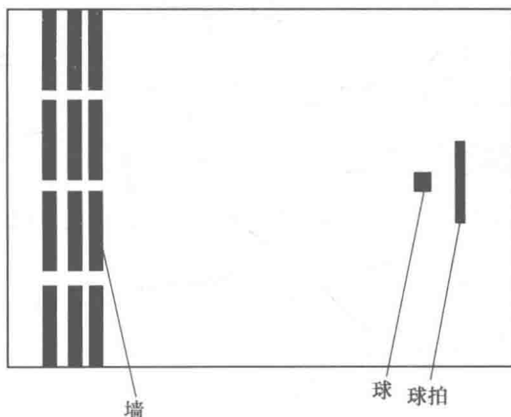


图 13-11　越狱游戏的屏幕

13.7.8　全屏圆点轨迹

我们可以使用外部 SRAM 芯片实现 13.5 节中的全屏圆点轨迹电路。其步骤如下：

1）修改第 11 章中 SRAM 控制部分，配置 SRAM 芯片为 $2^{19}\times 8$ 的存储器；

2）根据 13.5.2 节的讨论，在电路中包含新的存储模块。注意：访问外部存储器需要两个时钟周期；

3）综合并验证电路操作的正确性。

13.7.9　鼠标指针电路

鼠标接口在 10.5 节中已介绍。鼠标指针电路使用一个鼠标在屏幕中控制 16 ×16 的立方体的运动。其操作如下：

1）立方体依据鼠标的运动而运动；

2）当到达边界时，指针被包裹住；

3）当鼠标左键按下时指针改变颜色。改变的颜色是在表 13-1 中的 8 种颜色中循环的。

综合和验证电路操作的正确性。

13.7.10　小屏幕内鼠标轨迹电路

鼠标轨迹电路是要在 128 ×128 的屏幕中跟踪鼠标的轨迹，和 13.5 节中介绍的圆点轨迹类似，介绍如下：

- 3 位开关决定轨迹的颜色；
- 交替地点击鼠标左键表示鼠标按下和松开；
- 单击鼠标右键清除屏幕。

综合并验证电路操作的正确性。

13.7.11 全屏幕鼠标轨迹电路

重复 13.7.10 节中的试验，但是使用全屏。在这个电路中需要一个与 13.7.8 节中类似的 SRAM 存储器。

第 14 章　VGA 控制器Ⅱ：示例

14.1　简介

在第 13.3 节中论述了一个分布映射像素生成方案。一个点阵可被认为是一个“超级像素”。尽管在一个分布映射图案中一个像素被定义为 3 个字节，但是一个点阵只能被映射到一个预定图案中。以下有一个方法创建一个本文显示论述点阵的特性和利用分布映射方案设计像素生成电路。我们在这一章节中讨论这种方法，并且将它应用到设计碰球游戏的得分和规则中去。

14.2　举例

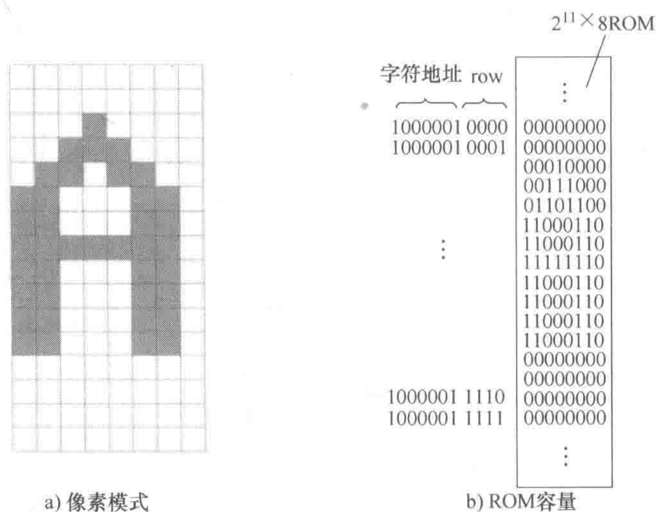
14.2.1　点阵的特性

当应用一个分布映射的方案的时候，我们视每个字符为一块点阵。在一个按位映射方案中，一个像素的值表示一个 3bit 图像。另一方面，一个点阵的值意味着特定形式的代码。为了实现文本显示，我们使用 7 位 ASCII 码描述点阵特性。

点阵的样式制定字体特性设置。这里有多种字形可供使用。我们选择一个 8 ×16 字型，就像在早期的 IBM 个人计算机中用的一样。在这一个字形中，每个字符被一个 8 ×16 字型样式所表现。如图 14-1a 中字母“A”模板所示。

字符类型被存储在一个 ROM 中每种类型需要 24 ×8bit 空间。字体 ROM 作为一种类型存储器被我们所熟知。最初的字形组有 256 种类型，包括数字、大小写字母、标点符号和许多特殊字符。我们只实现一半的样式而且排除大多数的特殊符号。通过计算，至少需要 $2^7 \times 2^4 \times 8$bitROM 的空间。它通常配置成一个 $2^{11} \times 8$ 的 ROM。

当我们在一个 640 ×480 的固定屏幕中使用这些 8 ×16 字符的时候，80 个点阵被用于匹配横向，30 个点阵被用于匹配纵向。在其他的字体中，屏幕可以被处理成一个 80 ×25 的点阵屏。我们可以把字符放置在这个用缩放坐标的屏幕上。



a) 像素模式　　b) ROM容量

图 14-1　字母 A 的字型模式

14.2.2　字体 ROM

我们的字体工具能设置 128 个 ASCII 码，如表 8-1 所示。这 128 个字符类型可以由一个 $2^{11}\times8$ 的字体 ROM 提供。一个 ROM 中有 7 个 11bit 地址的 MSB 被用来识别字符，有 4 个 LSB 的地址被用来识别一个字符模块占用几排。字母“A”的地址和 ROM 内容如图 14-1b 所示。

在 ASCII 表中，第一个栏由非可印刷控制字符组成。字体 ROM 使用这些代码组实现特别的图形符号。例如，06_{16}码将会在荧屏上产生一个黑桃图案“♠”。注意：为 00_{16}码保留一块空白的点阵。

2^{11}行 8 排字体 ROM 能很好地应用在 Spartan-3 的单口 block RAM 中。我们使用示例 12.6 的 ROM 模板确保一个 block RAM 将会在综合的时候被推断出来。如示例 14.1HDL 码所示。完整的代码有 2^{11}行，包含相应的注释，文件能从合作网站上下载。

示例 14.1　字体 ROM 的部分代码

```
module font_rom
  (
  input wire clk,
  input wire[10:0] addr,
  output reg[7:0] data
  );
```

```
// 信号声明
reg[10:01 addr_reg;
// 主体
always@ (posedge clk)
addr_reg <= addr;
always@ *
case( addr_reg)
  //x00 代码模块
  11'h000 : data = 8'b00000000;//
  11'h001 : data = 8'b00000000;//
  11'h002 : data = 8'b00000000;//
  11'h003 : data = 8'b00000000;//
  11'h004 : data = 8'b00000000;//
  11'h005 : data = 8'b00000000;//
  11'h006 : data = 8'b00000000;//
  11'h007 : data = 8'b00000000;//
  11'h008 : data = 8'b00000000;//
  11'h009 : data = 8'b00000000;//
  11'h00a : data = 8'b00000000;//
  11'h00b : data = 8'b00000000;//
  11'h00c : data = 8'b00000000;//
  11'h00d : data = 8'b00000000;//
  11'h00e : data = 8'b00000000;//
  11'h00f : data = 8'b00000000;//
  //x01 微笑表情代码
  11'h010 : data = 8'b00000000;//
  11'h011 : data = 8'b00000000;//
  11'h012 : data = 8'b01111110;//  ******
  11'h013 : data = 8'b10000001;// *      *
  11'h014 : data = 8'b10100101;// * *  * *
  11'h015 : data = 8'b10000001;// *      *
  11'h016 : data = 8'b10000001;// *      *
  11'h017 : data = 8'b10111101;// * **** *
  11'h018 : data = 8'b10011001;// *  **  *
  11'h019 : data = 8'b10000001;// *      *
```

```
      11'h01a : data = 8'b10000001;// *      *
      11'h01b : data = 8'b01111110;//  ******
      11'h01c : data = 8'b00000000;//
      11'h01d : data = 8'b00000000;//
      11'h01e : data = 8'b00000000;//
      11'h01f : data = 8'b00000000;//
      ...
      //x7f 代码
      11'h7f0 : data = 8'b00000000;//
      11'h7f1 : data = 8'b00000000;//
      11'h7f2 : data = 8'b00000000;//
      11'h7f3 : data = 8'b00000000;//
      11'h7f4 : data = 8'b00010000;//    *
      11'h7f5 : data = 8'b00111000;//   ***
      11'h7f6 : data = 8'b01101100;//  ** **
      11'h7f7 : data = 8'b11000110;// **   **
      11'h7f8 : data = 8'b11000110;// **   **
      11'h7f9 : data = 8'b11000110;// **   **
      11'h7fa : data = 8'b11111110;// *******
      11'h7fb : data = 8'b00000000;//
      11'h7fc : data = 8'b00000000;//
      11'h7fd : data = 8'b00000000;//
      11'h7fe : data = 8'b00000000;//
      11'h7ff : data = 8'b00000000;//
   endcase
endmodule
```

注意：以 block RAM 为基础的 ROM 执行输入需要一个时钟周期的延时，如第 12.4.3 节所讨论。

14.2.3　基本文本生成电路

依照目前的像素坐标（pixel_x 和 pixel_y）、外部的数据和控制信号，像素生成电路产生像素数值。像素建立在一个分布映射的系统的基础上，包含 2 个阶段。第一阶段使用信号 pixel_x 和 pixel_y 的高位生成点阵的代码，第二阶段使用第一阶段的代码和低位产生像素的值。

文本生成电路依据这个方法进行代码设计，基本图如图 14-2 所示。屏幕被制作成一个 80 * 30 的点阵，每个点阵包含一个 8 * 16 的字体。在第一阶段，信号 pixel_x [9：3] 和 pixel_y [8：4] 提供目前点阵位置的 *x* 和 *y* 坐标。字符生成电路使用这些坐标和其他的外部数据相结合，产生这一块点阵（标注：char_addr）的数值，其数值符合一个 ASCII 码字符。在第二个阶段，ASCII 码变成字体 ROM 的 7 个 MSB 地址，并详细说明了目前图案的位置。连接荧屏 *y* 坐标（pixel_y [3：0]，标注：row_addr）的 4 个 LSB 构成字体 ROM 全部的地址（标注：rom_addr）。在图案中字体 ROM（标注：font_word）的输出形式为一行 8 位。荧屏 *x* 坐标的 3 个 LSB（pixel_x [2：0]，标注：bit_addr）详细说明了要求的像素位置和 8 选 1 多路图像的输出。

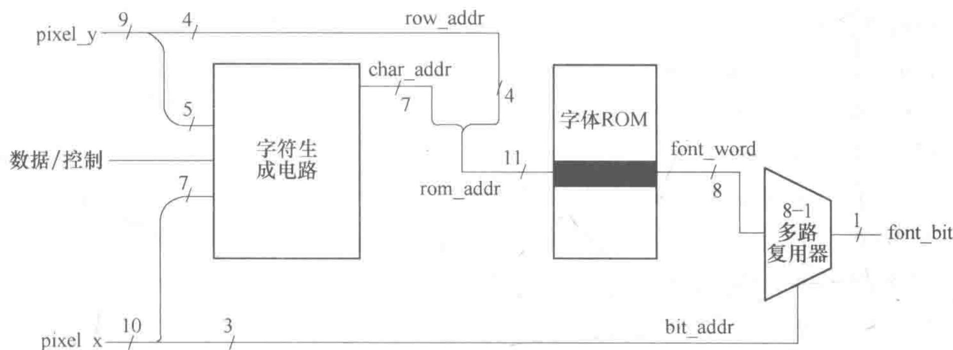


图 14-2 二级文本生成电路

14.2.4 字体显示电路

我们利用一个简单的字体显示电路验证字体 ROM 的功能，并在荧屏上显示所有的字体图案。128 个图案分布排列在 4 行中，图案如表 8-1 中的四列 ASCII 码表所示。通过使用适当的 *x* 和 *y* 坐标产生想得到的 ASCII 码，我们能获得每个图案。ASCII 码由信号 char_addr 表示。代码片段如下：

```
assign char_addr = {pixel_y[5:4],pixel_x[7:3]};
```

信号 pixel_x[7：3]形成 ASCII 码的 5 个 LSB，如此 32(2^5)个连续的字体图案将会被显示成一排。信号 pixel_y[5：4]形成 ASCII 码的 2 个 MSB，如此 4 个连续的列将会被显示。因为信号 pixel_y 和 pixel_x 高位溢出不明确，所以在荧屏上 32×4 区域被用来重复显示。一个另外的代码片段被包括在内只为荧屏的顶端左边部分打开显示。完整的代码如示例 14.2 所示。

示例 14.2　字型显示装置电路的图素生成代码

```
module font_test_gen
  (
  input wire clk,
  input wire video_on,
  input wire [9:0] pixel_x,pixel_y,
  output reg [2:0]rgb_text
  );
  //信号声明
  wire[10:0] rom_addr;
  wire[6:0] char_addr;
  wire[3:0] row_addr;
  wire[2:0]bit_addr;
  wire[7:0]font_word;
  wire font_bit,text_bit_on;
  //实体
  //示例字体ROM
  font_rom font_unit
    (.clk(clk),.addr(rom_addr),.data(font_word));
    //字体ROM 界面
    assign char_addr = {pixel_y[5:4],pixel_x[7:3]};
    assign row_addr = pixel_y[3:0];
    assign rom_addr = {char_addr ,row_addr};
    assign bit_addr = pixel_x[2:0];
    assign font_bit = font_word[ ~bit_addr];
    //"on"后缀限制在顶层区域使用
    Assign text_bit_on = (pixel_x[9:8] ==0 && pixel_y[9:6] ==0) ? font_bit :
1'b0;
    //rgb 多路电路
    always@ *
      if( ~video_on)
        rgb_text =3'b000; //空白
      else
      if(text_bit_on)
```

```
    rgb_text = 3'b010; // 绿色
  else
    rgb_text = 3'b000; // 黑色
endmodule
```

代码的主要部分是字体 ROM 界面。为了清晰地理解，我们在下面对字体 ROM 的信号给出了相应的定义，如图 14-2 所示。

- char_addr：7 位，字符的 ASCII 码；
- row_addr：4 位，在一个详细的字体图案中的行数；
- rom_addr：11 位，字体 ROM 地址；由 char_addr 和 row_addr 拼接；
- bit_addr：3 位，在一个详细的字体图案中的列数；
- font_word：8 位，由 rom_addr 指定的一个图素的行字体图案；
- font_bit：1 位，被 bit_addr 指定的 font_word 的一个像素。

这些信号根据图 14-2 所示的框图进行连接。font_bit 信号的流程由多路选择器完成，其代码如下：

```
assign font_bit = font_word[~bit_addr];
```

注意：在字体 ROM 中一行（也就是一个字）根据递减的顺序被定义（也就是[7：0]）。因为荧屏的 x 坐标于上升的方式被定义，数目从左边到右边增加，重新获得位的顺序必须取反。在语句中由“~”运算符完成。

我们需要结合同步电路和创建顶层描述。其 HDL 代码如示例 14.3 所示。

示例 14.3　字体显示电路顶层描述

```
module font_test_top
  (
   input wire clk, reset,
   output wire hsync, vsync,
   output wire [2:0] rgb
  );
   // 信号声明
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick;
   reg [2:0] rgb_reg;
   wire [2:0] rgb_next;
   // 实体
   // 示例: VGA 同步电路
```

```
vga_sync vsync_unit
 (.clk (clk),. reset (reset),. hsync (hsync),. vsync (vsync),. video_on
(video_on),. p_tick (pixel_tick),. pixel_x (pixel_x),. pixel_y (pixel_y));
//字体生成电路
font_test_gen font_gen_unit
 (.clk (clk),. video_on (video_on),. pixel_x (pixel_x),. pixel_y (pixel
_y), rgb_text (rgb_next));
//rgb 缓冲器
always@ (posedge clk)
if (pixel_tick)
rgb_reg <= rgb_next;
// 输出
Assign rgb = rgb_reg;
endmodule
```

在这个电路里存在微妙的时序问题。因为 block RAM 执行字体 ROM 输出时将经历一个时钟周期的延时。然而，因为信号 pixel_tick 每两个时钟周期会被判断一次，所以在信号 pixel_x 的数据未改变的时间间隔内，通信的数据（font_bit）会完全地被重新得到。多路 rgb 电路可以使用这些数据，并且采取及时的方式把期望值存储在寄存器 rgb_reg 中。

14.2.5 字体缩放比例

在分布映射的系统中，我们能依据一个点阵图案的比例通过“扩大”荧屏图素扩大尺寸。例如，我们可以依据比例把 8 * 16 字体扩大像素 4 次到 16 * 32 字体（也就是说，1 像素扩大到 4 像素）。为了执行缩放比例，我们仅仅需要把图素坐标向右移动 1 个位，而且丢弃信号 pixel_x 和 pixel_y 的 LSB。有一个例子能很好地解释。让我们在先前字体的地方重复显示扩大后的 16 * 32 字体。荧屏现在能被当做一个 40 * 15 点阵。新的字体地址会变成如下：

```
assign row_addr = pixel_y[4:1];
assign bit_addr = pixel_x[3:1];
assign char_addr = {pixel_y[6:5],pixel_x[8:4]};
```

首先这两个声明意味着当 pixel_x ［0］ 和 pixel_y ［0］ 为“00”、“01”、“10”和“11”的时候，相同的 font_bit 值将被获得，通过这个原始像素会有效地被扩大到 4 倍像素。text_bit_on 条件也需要被修改来适应一个更大的区域：

```
Assign text_bit_on = (pixel_x[9] ==0 && pixel_y[9:7] ==0)? Font_bit : 1'
```

b0;

我们能应用这个系统在字体上更进一步地扩展比例范围。注意：因为它们只是简单地放大最初的图案，而且没有引进更新的细节，所以被扩大的字体可能显现出锯齿形状。

14.3 全屏文本显示

正如其名称所说，全屏文本显示使用整个荧屏显示文本字符。现在字符生成电路包含了一个点阵存储器，它能存储每个点阵的 ASCII 码。点阵存储器的设计与第 13.5 节的位映射电路的视频存储器类似。为了使存储器容易存取，我们可以通过连接一个点阵的 *x* 和 *y* 坐标形成地址。这将转化为 12bit 的 80 * 30（也就是 27 * 25）点阵荧屏。因为每个点阵包含一个 7 位 ASCII 码，所以它需要一个 $2^{12}*7$ 的存储模块。一个同步双端口 RAM 能实现该目的。点阵存储电路如图 14-3 所示。

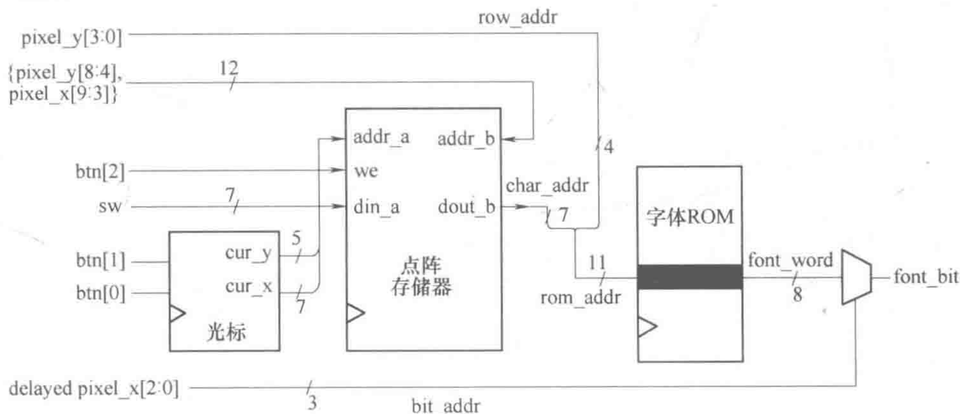


图 14-3　含点阵存储器的文本生成电路

因为访问点阵存储器需要另外的一个时钟周期，但是现在取回一个字体图案的时间增加到 2 个时钟周期。延长延时引起了一个敏感时序问题。因为信号 pixel_x 的更新需要两个时钟周期，所以当 font_word 变成有效时它的值会增加。因而，当数据通过声明被重新使用的时候，bit_addr 的增量被使用，而且一个不正确的字体数据将会被选择和发送输出。通过把信号 pixel_x 穿过两个 buffer 的方式来解决这个时序问题，用这个延时信号来代替 pixel_x 信号。

```
assign bit_addr = pix_x2_reg[2:0];
assign font_bit = font_word[~bit_addr];
```

我们使用一个简单的电路证明全荧屏分布映射系统的设计。电路从一个 7 位

开关器读取 ASCII 码，放置在 80 * 30 点阵荧屏上的显著位置。概要图如图 14-3 所示。光标进入到当前的位置时，其颜色会翻转。光标块保持光标当前位置的路径。电路使用 3 个按钮开关作为控制。其中两个按钮分别控制光标向右和向下移动。第三个按钮控制写操作。当按钮被按下的时候，7 位开关的当前数值被写到点阵存储器中。HDL 代码如示例 14.4 所示。

示例 14.4　全屏文本显示生成电路

```
module text_screen_gen
  (
  input wire clk,reset,
  input wire video_on,
  input wire[2:0]btn,
  input wire[6:0] sw,
  input wire[9:0] pixel_x,pixel_y,
  output reg[2 :0] text_rgb
  );
  // 信号声明
  // 字体ROM
  wire[10:0] rom_addr;
  wire[6:0] char_addr;
  wire[3:0] row_addr;
  wire[2:0] bit_addr;
  wire[7:0] font_word;
  wire font_bit;
  // 点阵RAM
  wire we;
  wire[11:0]addr_r,addr_w;
  wire[6:0] din,dout;
  //80 *30 点阵地图
  localparam MAX_X =80;
  localparam MAX_Y =30;
  // 光标
  reg[6:0] cur_x_reg;
  wire[6:0] cur_x_next;
  reg[4:0] cur_y_reg;
```

```
wire[4:0] cur_y_next ;
wire move_x_tick, move_y_tick, cursor_on ;
//像素计算延时
reg[9:0] pix_x1_reg,pix_y1_reg;
reg[9:0] pix_x2_reg,pix_y2_reg;
//输出信号
Wire[2 :0] font_rgb,font_rev_rgb ;
//实体
//2 个按钮的反弹电路实例
debounce deb_unit0
(.clk(clk),.reset(reset),.sw(btn[0]),.db_level(),.db_tick(move_x_tick));
debounce deb_unit1
(.clk(clk),.reset(reset),.sw(btn[1]),.db_level(),.db_tick(move_y_tick));
//例化字体ROM
font_rom font_unit
(.clk(clk),.addr(rom_addr),.data(font_word)) ;
//例化双口视频RAM (2^12 *7 )
Xilinx_dual_port_ram_sync
#(.ADDR_WIDTH(12),.DATA_WIDTH(7)) video_ram
(.clk(clk),.we(we),.addr_a(addr_w),.addr_b(addr_r),.din_a(),.dout_b
(dout));
//寄存器
always@(posedge clk)
  begin
    cur_x_reg <= cur_x_next;
    cur_y_reg <= cur_y_next;
  pix_x1_reg <= pixel_x;
  pix_x2_reg <= pix_x1_reg ;
  pix_y1_reg <= pixel_y ;
  pix_y2_reg <= pix_y1_reg ;
  end
  //写点阵RAM
  assign addr_w = {cur_y_reg,cur_x_reg};
  assign we = btn[2];
  assign din = sw;
```

```
// 读点阵RAM
// 使用非延迟整合形成的RAM 地址
assign addr_r = {pixel_y[8:4],pixel_x[9:3]};
assign char_addr = dout;
// 字体ROM
assign row_addr = pixel_y[3:0];
assign rom_addr = {char_addr,row_addr};
// 使用延迟整合选择一个位
assign bit_addr = pix_x2_reg[2:0];
assign font_bit = font_word[ ~bit_addr];
// 新光标位置
assign cur_x_next = (move_x_tick && (cur_x_reg == MAX_X - 1)) ? 0:// 包
                    (move_x_tick) ? cur_x_reg  +1  : cur_x_reg;
Assign cur_y_next = (move_y_tick && (cur_x_reg == MAX_Y - 1)) ? 0:// 包
                    (move_y_tick) ? cur_y_reg  +1  : cur_y_reg;
// 对象信号
// 实现光标在黑色和绿色视频接口的逆向转换
assign font_rgb = (font_bit) ? 3'b000 : 3'b000 ;
assign font_rev_rgb = (font_bit) ? 3'b000 : 3'b010 ;
// 为了对照使用延迟整合
assign cursor_on = (pix_y2_reg[8 :4] == cur_y_reg) &&
                   (pix_x2_reg[9 :3] == cur_x_reg);
//rgb 复用电路
always @ *
if( ~video_on)
tect_rgb = 3'b000; // 空白
else
if(cursor_on)
text_rgb = font_rev_rgb;
else
text_rgb = font_rgb ;
endmodule
```

字体 ROM 的界面信号类似于示例 14.2 中的信号，除去 char_addr 从点阵存储器读取端口获得的信号。为了使当前 x 和 y 坐标 pixel_x 和 pixel_y 能有效地存储在字体 ROM 中，我们创建了两个延时信号 pix_x2_reg 和 pix_y2_reg。注意：非延时信号 pixel_x 和 pixel_y 用来当做地址连接到字体 ROM 上，延时信号 pix_x2_reg 用来获取字体数据。双端口点阵 RAM 示例和界面与示例 13.7 中的视频 RAM 相似。

信号 cursor_on 被用来识别指针当前的位置。指针所在位置的字体图案的颜色被翻转。因为字体数据的两个时钟延时，所以我们使用延时坐标信号 pix_x2_reg 和 pix_y2_reg 相比较。

字体数据延时也为最终的信号 rgb 引入一个像素延时。这意味着一部分完全可见的 VGA 监视器向右移动了一个像素。为了解决这个问题，我们应修改 vga_sync 电路和使用延时信号 pix_x2_reg 和 pix_y2_reg 产生信号 hsync（水平同步）和 vsync（垂直同步）。当移动对全部视频效果影响不大时，我们就不做相应的修改。

顶层代码例化了文本像素生成电路和同步电路，如示例 14.5 所示。

示例 14.5　全屏文本显示顶层设计

```
module text_screen_top
  (
  input wire clk, reset,
  input wire[2:0]btn,
  input wire[6:0]sw,
  output wire hsync, vsync,
  output wire[2:0]rgb
  );
  // 信号声明
  wire[9:01 pixel_x,pixel_y;
  wire video_on,pixel_tick;
  reg[2:0]rgb_reg;
  wire[2:0] rgb_next;
  // 实体
  //vga 同步电路实例
  vga_sync vsync_unit
  (.clk(clk),.reset(reset),.hsync(hsync),.vsync(vsync),.video_on(video_on),
  .p_tick(pixe1_tick),.pixel_x(pixe1_x),.pixel_y(pixe1_y));
```

```
  // 字体生成电路
  text_screen_gen text_gen_unit
  (.clk(clk),.reset(reset),.video_on(video_on),.btn(btn),.sw(sw),
  .pixel_x(pixel_x),.pixel_y(pixel_y),.text_rgb(rgb_next));
  //RGB 缓冲器
  always@ (posedge clk)
  if(pixel_tick)
  rgb_reg <= rgb_next ;
  // 输出
  assign rgb = rgb_reg;
endmodule
```

14.4 完整的乒乓游戏设计

在 13.4.3 节我们为乒乓游戏设计创建了自由运行的图形电路。在本节中，我们增加一个文本界面用于显示得分和信息，设计了一个整合了图形和文本子系统的顶层控制状态机来配置整个电路系统的运行。完整的乒乓游戏的规则和操作流程如下：

- 当游戏开始的时候，显示文本规则；
- 接下来玩家按下按钮，游戏开始；
- 玩家记下每次用球拍击球的分数；
- 当玩家丢球的时候，游戏中止，重新发球，每个玩家拥有 3 个球；
- 得分和剩余的球数在屏幕上方显示；
- 当玩家丢掉 3 个球之后，游戏结束，显示游戏结束信息。

在下面的部分，我们首先讨论了文本子系统、图解子系统和辅助计数器，然后从顶层状态机到坐标和全部操作控制。概要图如图 14-4 所示。

14.4.1 文本子系统

乒乓游戏的子系统由 4 个文本信息组成：

- 在荧屏上面用 16 * 32 字体，使用“Scores：DD”和“Ball：D”方式分别显示得分和剩余球数；
- 在游戏开始的时候，使用常用的字体显示规则信息“Rules：Use two buttons to move paddle up or down”；
- 标识语“PONG”于 64 * 128 的字体在背景上显示；

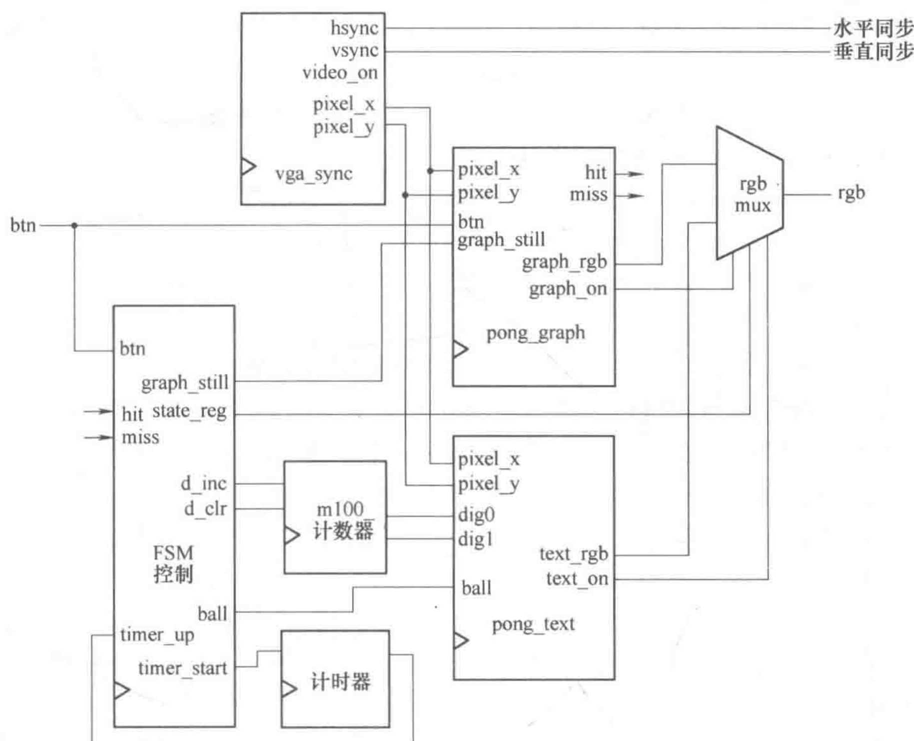


图 14-4　完整的乒乓游戏设计顶层模块图

- 在游戏结束后，于 32 * 64 的字体显示游戏结束提示信息 “Game Over”。

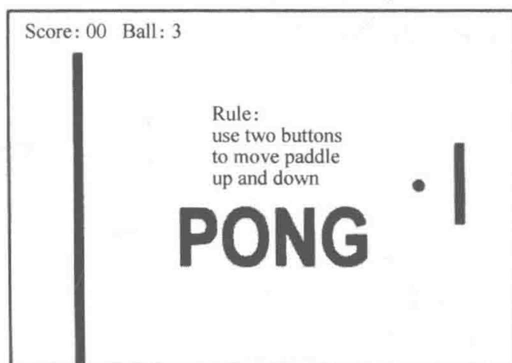


图 14-5　乒乓游戏文本

初始三条概要信息如图 14-5 所示。游戏结束后的信息在规则信息和不包括在内的信息之间相互交替变换。

由于那些信息使用不同的字体尺寸在不同的场合显示，所以我们不能在一个单独场景中创建那些信息。我们视每个文本信息都是一个单独对象并产生状况信号和字体 ROM 地址。例如，信息标识语片段如下：

```
assign logo_on = (pix_y[9:7] ==2) &&(3 <= pix_x[9:6]) && (pix_x[9:6]
<=6);
assign row_addr_1 = pix_y[6:3];
```

```
  assign bit_addr_l = pix_x[5:3];
  always @ *
    case (pix_x[8:6])
      3'o3: char_addr_l =7'h50; //P
      3'o4: char_addr_l =7'h4f; //O
      3'o5: char_addr_l =7'h4e; //N
    default: char_addr_l =7'h47; //G
  endcase
```

信号 logo_on 指出在标识语区域当前浏览的相应文本是“turned on”。其他声明详细说明了信息内容和字体 ROM 连接生成 32×64 的鳞形字符。其他三部分类似。单独的多路电路检查不同的信号和对字体 ROM 的地址的设置路径。

文本子系统通过接口 ball、dig0 和 dig1 接收得分和剩余球的数量。通过接口 rgb_text 输出 rgb 信息，通过 4 位接口 text_on 连接 4 个单独的信号输出状态信息。完整的代码如示例 14.6 所示。

示例 14.6　乒乓游戏的文本子系统

```
module pong_text
(
input wire clk,
input wire[1:0] ball,
input wire[3:0] dig0, dig1,
input wire[9:0] pix_x, pix_y,
output wire[3:0] text_on,
output reg[2:0] text_rgb
);
// 信号声明
wire[10:0] rom_addr;
reg[6:0] char_addr , char_addr_s , char_addr_l ,char_addr_r, char_addr_o;
reg[3:0] row_addr;
wire[3:0] row_addr_s , row_addr_l , row_addr_r , row_addr_o;
reg[2:0] bit_addr;
wire[2:0] bit_addr_s, bit_addr_l ,bit_addr_r, bit_addr_o;
wire[7:0] font_word;
wire font_bit , score_on, logo_on, rule_on, over_on;
wire[7:0] rule_rom_addr;
```

```
// 例化字体ROM
font_rom font_unit
(.clk(clk),.addr(rom_addr)..data(font_word));
//--------------------------------------
// 得分区
// -在左上角显示两位数的得分和剩余球数
// -使用16 *32 字体
// -第一行,16 个字符:"Score :DD Ball :D"
//--------------------------------------
assign score_on = (pix_y[9:5] ==0) && (pix_x[9:4] <16) ;
assign row_addr_s = pix_y[4:1];
assign bit_addr_s = pix_x[3:1];
always @ *
case(pix_x[7:4])
4'h0: char_addr_s =7'h53; //S
4'h1: char_addr_s =7'h63; //c
4'h2: char_addr_s =7'h6f; //o
4'h3: char_addr_s =7'h72; //r
4'h4: char_addr_s =7jh65; //e
4'h5: char_addr_s =7'h3a; // :
4'h6: char_addr_s = {3'b011, digl}; // 十位数字
4'h7: char_addr_s = {3'b011, dig0}; // 个位数字
4'h8: char_addr_s =7'h00; //
4'h9: char_addr_s =7'h00; //
4'ha: char_addr_s =7'h42; //B
4'hb: char_addr_s =7'h61; //a
4'hc: char_addr_s =7'h6c; //1
4'hd: char_addr_s =7'h6c; //1
4'he: char_addr_s =7'h3a; // :
4'hf: char_addr_s = {5'b01100, ball};
Endcase
//--------------------------------------
// 标识语区:
// -在顶部中间显示标识语"PONG"
// -作为背景使用
```

```
//  -使用64 *128 字体
//--------------------------------------
assign logo_on = (pix_y[9:7] ==2) &&(3 <=pix_x[9: 6] ) && (pi_x[9: 6] <=
6) ;
assign row_addr_1 = pix_y[6:3];
assign bit_addr_1 = pix_x[5:3];
always @ *
case(pix_x[8:6])
3'03: char_addr_1 =7'h50; //P
3'04: char_addr_1 =7'h4f; //0
3'05: char_addr_1 =7'h4e; //N
default :char_addr_1 =7'h47; //G
Endcase
//----------------------------
// 规则区
//  -在中央显示规则(4 *16 字体)
//  -规则文本:
//Rule  :
//use two buttons
//to move paddle
//up and down
//----------------------------
assign rule_on = (pix_x[9: 7] ==2) && (pix_y[9: 6] ==2);
assign row_addr_r = pix_y[3: 0];
assign bit_addr_r = pix_x[2: 0];
assign rule_rom_addr = {pix_y[5:4],pix_x[6:3]};
always @ *
case(rule_rom_addr)
//第1 排
                6'h00: char_addr_r =7'h52; //R
                6'h01: char_addr_r =7'h55; //U
                6'h02: char_addr_r =7'h4c; //L
                6'h03: char_addr_r =7'h45; //E
                6'h04: char_addr_r =7'h3a; // :
                6'h05: char_addr_r =7'h00; //
```

```
6'h06: char_addr_r = 7'h00; //
6'h07: char_addr_r = 7'h00; //
6'h08: char_addr_r = 7'h00; //
6'h09: char_addr_r = 7jh00; //
6'h0a: char_addr_r = 7'h00; //
6'h0b: char_addr_r = 7'h00; //
6'h0c: char_addr_r = 7'h00; //
6'h0d: char_addr_r = 7'h00; //
6'h0e: char_addr_r = 7'h00; //
6'h0f: char_addr_r = 7'h00; //
//第2 排
6'h10: char_addr_r = 7'h55; //U
6'h11: char_addr_r = 7'h73; //s
6'h12: char_addr_r = 7'h65; //e
6'h13: char_addr_r = 7'h00; //
6'h14: char_addr_r = 7'h74; //t
6'h15: char_addr_r = 7'h77; / /w
6'h16: char_addr_r = 7'h6f ; //o
6'h17: char_addr_r = 7'h00; //
6'h18: char_addr_r = 7'h62; //b
6'h19: char_addr_r = 7'h75; //u
6'h1a: char_addr_r = 7'h74; //t
6'h1b: char_addr_r = 7'h74; //t
6'h1c: char_addr_r = 7'h6f ; //o
6'h1d: char_addr_r = 7'h6e; //n
6'h1e: char_addr_r = 7'h73; //s
6'h1f: char_addr_r = 7'h00; //
//第3 排
6'h20: char_addr_r = 7'h74; //t
6'h21: char_addr_r = 7'h6f; //o
6'h22: char_addr_r = 7'h00; //
6'h23: char_addr_r = 7'h6d; //m
6'h24: char_addr_r = 7'h6f ; //o
6'h25: char_addr_r = 7'h76; //v
6'h26: char_addr_r = 7'h65; //e
```

```
        6'h27: char_addr_r = 7'h00; //
        6'h28: char_addr_r = 7'h70; //p
        6'h29: char_addr_r = 7'h61; //a
        6'h2a: char_addr_r = 7'h64; //d
        6'h2b: char_addr_r = 7'h64; //d
        6'h2c: char_addr_r = 7'h6c; //l
        6'h2d: char_addr_r = 7'h65; //e
        6'h2e : char_addr_r = 7'h00; //
        6'h2f: char_addr_r = 7'h00; //
        //第4 排
        6'h30: char_addr_r = 7'h75; //u
        6'h31: char_addr_r = 7'h70; //p
        6'h32: char_addr_r = 7'h00; //
        6'h33: char_addr_r = 7'h61; //a
        6'h34: char_addr_r = 7'h6e; //n
        6'h35: char_addr_r = 7'h64; //d
        6'h36: char_addr_r = 7'h00; //
        6'h37: char_addr_r = 7'h64; //d
        6'h38: char_addr_r = 7'h6f ; //o
        6'h39: char_addr_r = 7'h77; //w
        6'h3a: char_addr_r = 7jh6e; //n
        6'h3b: char_addr_r = 7'h2e; //
        6'h3c: char_addr_r = 7'h00; //
        6'h3d: char_addr_r = 7'h00; //
        6'h3e: char_addr_r = 7'h00; //
        6'h3f: char_addr_r = 7'h00; //
    endcase
//------------------------------------------------------------
// 游戏结束区
//  -在中央显示"Game Over"
//  -使用32 *64 字体
//------------------------------------------------------------
assign over_on = (pix_y[9: 6] ==3)&&(5 <= pix_x[9: 5] )&&(pix_x
                [9: 5] <=13);
assign row_addr_o = pix_y[5: 2] ;
```

```
assign bit_addr_o = pix_x[4:2] ;
always @ *
    case (pix_x[a: 5] )
        4'h5: char_addr_o = 7'h47; //G
        4'h6: char_addr_o = 7'h61; //a
        4'h7: char_addr_o = 7'h6d; //m
        4'h8: char_addr_o = 7'h65; //e
        4'h9: char_addr_o = 7'h00; //
        4'ha: char_addr_o = 7'h4f; //0
        4'hb: char_addr_o = 7'h76; //v
        4'hc: char_addr_o = 7'h65; //e
        default : char_addr_o = 7'h72; //r
    endcase
//----------------------------------------------------------
// 字体ROM 地址和RGB 多路选择器
//----------------------------------------------------------
always @ *
    begin
        text_rgb = 3'b110; // 背景,黄色
        if(score_on)
            begin
                char_addr = char_addr_s;
                row_addr = row_addr_s;
                bit_addr = bit_addr_s;
                if (font_bit)
                    text_rgb = 3'b001;
            end
        elseif(rule_on)
            begin
                char_addr = char_addr_r;
                row_addr = row_addr_r;
                bit_addr = bit_addr_r ;
                if (font_bit)
                    text_rgb = 3jb001;
            end
```

```
        elseif (logo_on)
            begin
                char_addr = char_addr_l;
                row_addr = row_addr_l;
                bit_addr = bit_addr_l;
                if (font_bit)
                    text_rgb = 3'b011;
            end
        else // 游戏结束
            begin
                char_addr = char_addr_o ;
                row_addr = row_addr_o;
                bit_addr = bit_addr_o;
                if (font_bit)
                    text_rgb = 3'b001;
            end
        end
assign text_on = (score_on, logo_on, rule_on, over_on};
//---------------------------------------------------
// 字体ROM 接口
assign rom_addr = (char_addr, row_addr};
assign font_bit = font_word[ ~bit_addr] ;
        endmodule
```

每部分的结构都是类似的。因为信息短，所以使用 ROM 模板规范对它们进行编码。此外没有使用时钟信号，推断应该是分布式 RAM 或组合逻辑。根据 2 个 4bit 的外部信号 dig0 和 dig1 产生 2 个数字存储器。注意数字 0，1，…，9 的 ASCII 编码是 3016，3116，…，3916。我们能简单地通过在 dig0 和 dig1 前连接“011”产生 char-addr 信号。

14.4.2 修正图像分系统

为了适应新的顶层控制器，13.4.3 节电路图要求做以下更改：

● 增加一个 gra-still 控制信号（为“still graphics”），当该信号被声明后，不用移动，垂直线被放置在中间，球被放置在屏幕中心；

● 增加 hit、miss 状态信号，当球拍击中球的时候，hit 信号会被置一个时钟

周期有效，当球拍没有击中球并且越过右边边界的时候，置 miss 信号有效；

- 增加一个 graph-on 信号用于显示图像分系统的状态。

代码的更改部分在示例 14.7 中显示。

示例 14.7　乒乓游戏图像分系统的更改部分

```
    ...
// 新的球位置
assign ball_x_next = (gra_still)  ?   MAX_X/2 :
                     (refr_tick)  ?   ball_x_reg + x_delta_reg :
                     ball_x_reg ;
assign ball_y_next = (gra_still)  ?   MAX_Y/2 :
                     (refr_tick)  ?   ball_y_reg + y_delta_reg :
                     ball_y_reg ;
// 新的球速度
always @ *
begin
    hit = 1'b0;
    miss = 1'b0;
    x_delta_next = x_delta_reg;
    y_delta_next = y_delta_reg;
    if (gra_still) // 初始速度
    begin
        x_delta_next = BALL_V_N ;
        y_delta_next = BALL_V_P;
    end
    elseif (ball_y_t < 1)  // 到达顶部
        y_delta_next = BALL_V_P;
    elseif (ball_y_b > (MAX_Y_1))  // 到达底部
        y_delta_next = BALL_V_N;
    elseif (ball_x_1 <= WALL_X_R) // 到达墙
        x_delta_next = BALL_V_P; // 弹回
    elseif ((BAR_X_L <= ball_x_r) && (ball_x_r <= BAR_X_R) &&
        (bar_y_t <= ball_y_b) && (ball_y_t <= bar_y_b))
    begin
    // 到达右边并且撞击,球弹回
```

```
        x_delta_next = BALL_V_N;
        hit = 1'b1;
    end
    elseif (ball_x_r > MAX_X) // 到达右边界
        is miss = 1'b1;   // 未击中
    end
    ...
    assign graph_on = wall_on | bar_on | rd_ball_on;
    ...
```

14.4.3 辅助计数器

为了方便计算，顶层设计要求有两个有用的小模块，m100-counter 和 timer。m100_counter 模块是一个两位的十进制计数器，计数范围从 00 ~ 99，用于记录比赛的得分。d_inc 和 d_clr 控制信号分别用于增加和清除该计数器。示例 14.8 显示了该部分代码。

示例 14.8　两位十进制计数器

```
module m100_counter
        (
            input wire clk, reset,
            input wire d_inc, d_clr,
            output wire[3:0] dig0, dig1
        );
    // 信号声明
    reg[3:0] dig0_reg, dig1_reg, dig0_next, dig1_next;
    // 寄存器
    always @ (posedge clk , posedge reset )
        if (reset)
            begin
                dig1_reg <= 0;
                dig0_reg <= 0;
            end
        else
            begin
```

```
            digl_reg <= digl_next ;
            dig0_reg <= dig0_next ;
        end
// 次态逻辑
always @ *
    begin
        dig0_next = dig0_reg ;
        digl_next = digl_reg;
        if (d_clr)
            begin
                dig0_next =0 ;
                digl_next =0 ;
            end
        elseif (d_inc)
            if (dig0_reg ==9)
                begin
                    dig0_next =0;
                    if (digl_reg ==9)
                      digl_next =0;
                    else
                      digl_next = digl_reg +1;
                    end
                else //dig0,不是9
                    dig0_next = dig0_reg +1;
        end
    //输出
    assign dig0 = dig0_reg;
    assign digl = digl_reg;
    endmodule
```

Timer 模块使用 60Hz 周期脉冲 timer-tick 来产生一段 2s 的时间间隔。它的目的是使视频在屏幕切换时暂停。当 timer-start 信号被声明，并且当 2s 的时间间隔到了，timer-up 信号有效时，该计数器开始计数。代码如示例 14.9 所示。

示例 14.9　2s 计数器

```
module timer
      (
         input wire clk, reset,
         input wire timer_start , timer_tick,
         output wire timer_up
      );
   // 信号声明
   Reg[6:0] timer_reg, timer_next;
   // 寄存器
   always @ (posedge clk, posedge reset)
      if (reset)
        timer_reg <=7'blllllll;
      else
        timer_reg <=timer_next ;
   // 次态逻辑
   always @ *
      if (timer_start)
        timer_next =7'blllllll;
      elseif ((timer_tick) && (timer_reg ! =0))
        timer_next =timer-reg - 1;
      else
        timer_next =timer_reg;
   // 输出
   assign timer_up =(timer_reg ==0);
   endmodule
```

14.4.4　顶层系统

乒乓游戏的顶层系统由先前的设计模块组成，包括视频同步电路、图像分系统、文本分系统和有效的计数器，以及有限状态机控制电路和 rgb 复用电路。其结构框图如图 14-4 所示。

状态机控制和监控整个系统的操作及文本和图像子系统的相关动作。其 ASMD 图如图 14-6 所示。状态机有下列 4 个状态和相关操作：

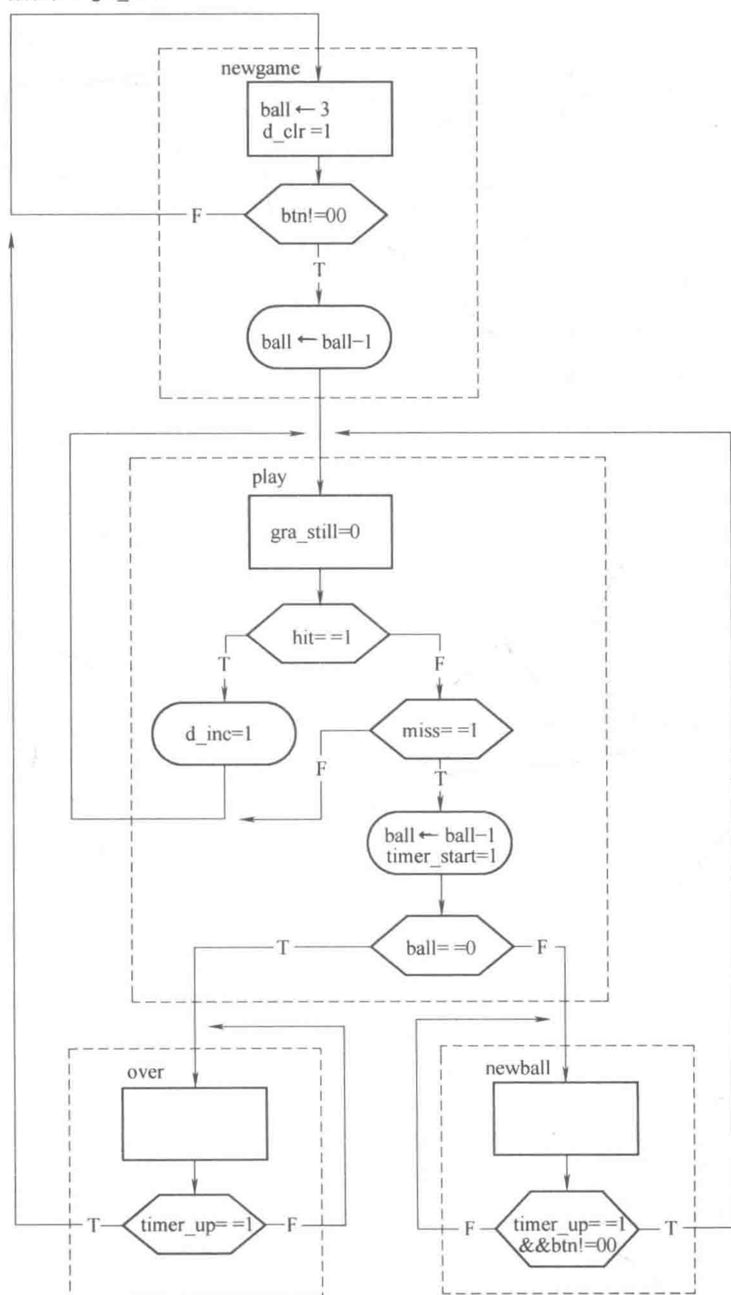


图 14-6　乒乓控制器 ASMD 图

● 最初状态机是在 newgame 状态，当按下一个按钮时游戏开始，同时状态机跳到 play 状态；

● 在 play 状态，状态机连续检验 hit 和 miss 信号，当 hit 信号有效时，为了在一个时钟周期使得分计数器加一，d_inc 信号被声明，当 miss 信号被声明时，FSM 激活 2s 计数器，ball 数量减一，同时检测剩下的球的数量，如果球数为零，游戏结束，同时状态机跳到 over 状态。否则状态机跳到 newball 状态；

● 在 newball 状态时，状态机等到 2s 计数间隔计满（如这时 timer-up 信号被申明），同时一个按钮被按下，接下来跳到 play 状态继续游戏；

● 状态机保持在 over 状态直到 2s 计数间隔计满，然后跳到 newgame 状态开始新的游戏。

rgb 复用电路依据 text-on 和 graphic-on 信号发送 text_rgb 或 graph_rgb 信号至输出。其关键部分代码如下：

```
always @ *
    if ( ~video_on)
        rgb_next = "000" ; // 空白边缘/折回
    else
    // 显示得分,规则,或者游戏结束
        if ( text_on[3] ||
                ((state_reg == newgame) && text_on[1]) ||
                ((state_reg == over) && text_on[0]) )
            rgb_next = text_rgb;
        elseif (graph_on) //display graph
            rgb_next = graph_rgb;
        elseif (text_on[2]) // 显示标志
            rgb_next = text_rgb;
        else
            rgb_next =3'b110; // 黄色背景
    // 输出
assign rgb = rgb_reg;
```

text-on [3] 信号是用于得分，会一直显示。text-on [1] 信号是用于规则，仅在状态机处于 newgame 状态时显示。同样，游戏结束信息的状态由 text-on [0] 信号显示，仅在状态机在 over 状态时显示。Logo 的状态由 text-on [2] 信号显示，它作为背景的一部分使用，仅在没有其他声明信号时显示。

完整代码如示例 14.10 所示。

示例 14.10 状态机游戏顶层系统

```
module pong_top
        (
            Input wire clk,reset,
            input wire[1:0]btn,
            output wire hsync,vsync,
            output wire[2:0]rgb
        );
   //符号状态声明
   localparam[1:0]
   newgame =2'b00,
   play =2'b01,
   newball =2'b10,
   over =2'bll;
  //信号声明
  reg[1:0] state_reg, state_next;
  wire[9:0] pixel_x, pixel_y ;
  wire video_on, pixel_tick, graph_on, hit, miss;
  wire[3:0] text_on;
  wire[2:0] graph_rgb, text_rgb ;
  reg[2:0] rgb_reg, rgb_next ;
  wire[3:0] dig0, dig1;
  reg gra_still, d_inc, d_clr, timer_start ;
  wire timer_tick, timer_up;
  reg[1:0] ball_reg , ball_next ;
  //----------------------------------------------------------
  //实例
  //----------------------------------------------------------
  //视频同步单元实例
  vga-sync vsync_unit
      (. clk(clk), . reset (reset), . hsync(hsync), . vsync(vs~nc),
        . video_on(video_on), . p_tick(pixe1_tick),
        . pixel_x(pixe1_x), . pixel_y(pixe1_y));
  //文本模块实例
```

```
pong_text text_unit
     (.clk (clk) ,
       .pix_x(pixe1_x),.pix_y(pixe1_y),
       .dig0(dig0),.digl(digl),.ball(ball_reg),
       .text_on(text_on),.text_rgb(text_rgb));
// 图形模块实例
pong_graph graph_unit
   (.clk(clk),.reset(reset),.btn(btn),
     .pix_x(pixel_x),.pix_y(pixel_y),
     .gra_still(gra_still),.hit(hit),.miss(miss),
     .graph_on(graph_on),.graph_rgb(graph_rgb));
//2s 计时器实例
//60 Hz tick
assign timer_tick = (pixel_x ==0) && ( pixel_y ==0);
timer timer_unit
     (.clk(clk),.reset(reset),.timer_tick(timer_tick),
       .timer_start(timer_start),.timer_up(timer_up));
//2 位十进制计数器实例
m100-counter counter_unit
     (.clk(c1k),.reset(reset),.d_inc(d_inc),.d_clr(d_clr),
       .dig0(dig0),.digl(digl));
//.......................................................
//FSMD
//.......................................................
//FSMD 状态& 数据寄存器
always @ (posedge clk,posedge reset)
     if(reset)
          begin
               state_reg <= newgame ;
               ball_reg <=0 ;
               rgb_reg <=0 ;
          end
     else
          begin
               state_reg <= state_next;
```

```
            ball_reg <= ball_next;
            if(pixel_tick)
                rgb_reg <= rgb_next ;
        end
//FSMD 次态逻辑
always @ *
    begin
        gra_still = 1'b1;
        timer_start = 1'b0;
        d_inc = 1'b0;
        d_clr = 1'b0;
        state_next = state_reg;
        ball_next = ball_reg;
        case(state_reg)
            newgame:
                begin
                    ball_next = 2'b11;// 三个球
                    d_clr = 1'b1;// 得分清零
                    if(btn!  =2b00)// 按下按钮
                        begin
                            state_next = play;
                            ball_next = ball_reg_1;
                        end
                end
            play:
                begin
                    gra_still = 1'b0; // 动画屏幕
                    if(hit)
                        d_inc = 1'b1; // 增加得分
                    elseif(miss)
                        begin
                            if(ball_reg ==0)
                                state_next = over;
                            else
                                state_next = newball;
```

```
                        timer_start = 1'b1;//2s 计时器
                        ball_next = ball_reg_1;
                    end
                end
            newball :
            // 等待2s 直到按下按钮
                if(timer_up&&(btn! =2'b00))
                    state_next = play;
            over:
            // 等待2s 显示游戏结束
                if(timer_up)
                    state_next = newgame ;
        endcase
    end
//..............................................................
//RGB 复用电路
//..............................................................
always @ *
if( ~video_on)
    rgb_next = "000" ;// 空白边缘/ 折回
else
// 显示得分,规则,或者游戏结束
    if(text_on[3] ||
        ((state_reg == newgame) && text_on[1]|| // 规则
        ((state_reg == over) && text_on[0])
        rgb_next = text_rgb;
    elseif(graph_on) // 显示图形
        rgb_next = graph_rgb;
    elseif(text_on[2]) // 显示标志
        rgb_next = text_rgb;
    else
        rgb_next = 3'b110; // 黄色背景
    // 输出
 assign rgb = rgb_reg;
endmodule
```

14.5　文献备注

可以用到几种不同特征的字体。Rapid Prototyping of Digital Systems 由 James. O. Hamblen 等使用压缩的64字符 8 * 8 字体设置。分片映射图像对文本显示没有限制，在早期的视频游戏中广泛使用。Steven Collins（ACMSIGGRAPH, May 1998）的文章《8位计算机游戏时代期间的计算机制图》全面地回顾了基于分片模式游戏的历史和设计技术。

14.6　实验

14.6.1　旋转旗帜

旋转旗帜在监控屏幕从右到左移动和旋转。它类似于 Windows 的 Marquee 屏幕保护。让旗帜上的文本是“Hello, FPGA World”。标语应该以四种不同字体大小显示，而且能以四种不同的速度传播。字体的大小和速度由4个开关控制。编写 HDL 描述，然后综合，最后验证电路的功能。

14.6.2　指针的下划线

在14.3节中，全屏文本显示电路使用反色显示当前指针的位置。将设计更改为使用下划线来显示指针位置。编写 HDL 描述，然后综合，最后验证电路功能。

14.6.3　双模式文本显示

有时对文本最好在屏幕上垂直显示，这能够通过旋转监控器90°然后停留在它的一侧来实现。按以下方法设计电路：

1）在14.3节中将全屏文本显示更改为垂直屏幕显示；

2）为了创造“双模式”文本显示，将水平和垂直设计合并，使用一个开关来选择期望的模式；

3）编写 HDL 描述，然后综合，最后验证电路功能。

14.6.4　键盘文本输入

代替开关和按钮，更自然的是使用键盘文本输入。我们可以使用4个箭头键来移动指针，使用常规键输入字符。使用键盘界面设计新电路已在9.4节讨论。编写 HDL 描述，然后综合，最后验证电路功能。

14.6.5 UART 终端

UART 终端收到来自 UART 端口的输入，然后在监控器上显示收到的字符。当连接 PC 串口时，应该在 Window 超级终端回放该文本。具体细节如下：

- 指针用来显示当前的位置；
- 当“carriage return”被赋值 $0d_{16}$时，屏幕开始一条新的线路；
- 当输入到 80 个字符后该行跳转（即，开始新的行）；
- 当指针到达屏幕的底部时（即，最后一行），第一行将被丢弃，所有其他行将会上提一个位置（即，scroll up）。

编写 HDL 描述，然后综合，最后验证电路功能。

14.6.6 方波显示

通过利用图 14-7a 中显示的 4 个简单的片段图案，我们能绘制一个方波。下面是 14.3 节中全屏文本的显示过程，目的是设计全屏波形编辑器：

1）设置片段的尺寸是 8 × 64，为 4 副图案创建 ROM 区；

2）计算在 640 ×480 分辨率的屏幕上片段的数量，同时为片段存储分配合适的配置；

3）使用 3 个控制按钮和 2 位用于模式进入的开关；

4）编写 HDL 描述，然后综合，最后验证电路功能。

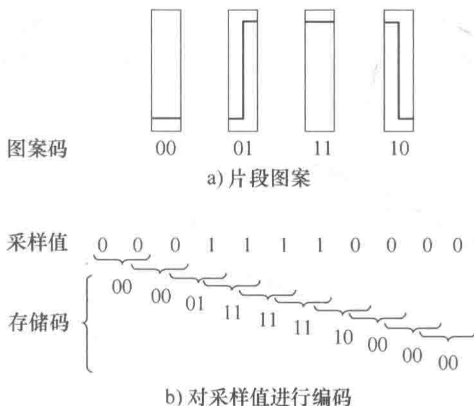


图 14-7　方波的片段图案和编码

14.6.7 简单的四路逻辑分析器

逻辑分析器显示了数字信号集的波形。我们想要设计一个简单的逻辑分析器，它能采样四输入信号在“自由航行”模式下的波形。代替使用触发模式，按钮开关被激活时开始数据采样。简单地，我们假设输入波形的频率范围在 10 ~100kHz。电路可以按下面的方法来设计：

1）使用采样周期脉冲来采样四输入信号，确保选择合适的速率，以便期望的输入频率范围可以在屏幕上完全显示；

2）对于采样信号点，它的值能被编码成包含前一点值的片段模式，例如，如果一个信号的采样顺利是“0000 1111 000”，片段模式变成“00 00 00 01 11 11 11 10 00 00”，如图 14-7b 所示；

3）按照前述方波实验过程来设计片段存储器和为显示 4 个存储波形的视频界面；

4）编写 HDL 描述，然后综合，最后验证电路功能。

为了验证电路功能，我们能通过前端的原型板连接 4 个外部信号。或者，我们可以设计一个顶层测试模型，它包括一个 4bit 的计数器（即，大约 50 kHz 的十进制计数器）和逻辑分析器，重新综合电路，验证其功能。

14.6.8 完整的双人乒乓游戏

在 13.7.6 节的实验描述了自由运行式双人乒乓游戏。按照 14.4 节乒乓游戏的过程来设计整个系统。这个系统设计应该包括一个新文本显示分系统的设计和顶层有限状态机控制器的设计。编写 HDL 语言描述设计，然后综合，最后验证电路功能。

14.6.9 完整的通关游戏

在 13.7.7 节的实验中描述了自由运行式通关游戏。按照 14.4 节乒乓游戏的过程来设计整个系统。应该包括一个新文本显示分系统设计和顶层有限状态机控制器的设计。编写 HDL 语言描述设计，然后综合，最后验证电路功能。

第三部分

PicoBlaze 微控制器

第 15 章　PicoBlaze 概述

15.1　简介

PicoBlaze 处理器是 Xilinx FPGA 芯片的一种紧凑的 8 位微控制器核。它是由单元级 HDL 描述（即软核），可以伴随其他的逻辑而综合。PicoBlaze 最大的优势是效率高，而且仅占用大约 200 个逻辑单元，少于 3S200 器件资源的 5%。PicoBlaze 没有被设计成一种高性能处理器，它具有很好的紧凑性和灵活性，可以用来进行简单的数据处理和控制，尤其是用作非时序的看门狗（housekeeping）和 I/O 操作。PicoBlaze 处理器可以很方便地组合成一个较大的系统和增强基于 FPGA 的设计的灵活性。

尽管具体的汇编程序以及微控制器的详细描述在本书中没有涉及，但该章节对 PicoBlaze 的组织架构以及指令集进行了全面地阐述，同时通过一系列具体的例子说明了如何使用汇编语言进行开发和描述 I/O 接口。另外，本章回顾了 PicoBlaze 的组织架构以及指令集，在 16 章中介绍了汇编语言编程，在 17 章和 18 章中讨论了通用的 I/O 接口和中断接口。

15.2　定制硬件和软件

15.2.1　从专用 FSMD 到通用微控制器

RTL 级设计以及在第 6 章中介绍的 FSMD 为将一个连续的算法转化成定制的硬件提供了通用的方法。重新排列的方块图见图 15-1a。在 FSMD 中，所有的组成部分，包括寄存器数量、寄存器的输入/输出路径、函数单元的数量和类型以及 FSM 的控制部分都被根据特定的应用而排列。如图所示，数据路径可能包含多种的函数单元和传输路径。

对于不同的应用，一种折中的方法就是保持硬件不变而使用定制的软件。转化的方式如下所述。首先，如图 15-1b 顶层所示，使用一种特定的结构来替换定制的数据路径。数据寄存器和定制的传输网络由一个寄存器文件来替换，这一文件具有特定数量的寄存器以及只包含两个读端口和一个写端口。而定制的函数单元由一个 ALU（算术逻辑单元）替换，且 ALU 只能执行一系列预先定义的函

数。这样，数据路径只能按照下面的格式来执行代码：

$$rd \leftarrow r1 \text{ op } r2$$

式中，r1、r2 和 rd 表示两个源寄存器和一个目的寄存器的地址，op 表示一种可用的 ALU 函数。

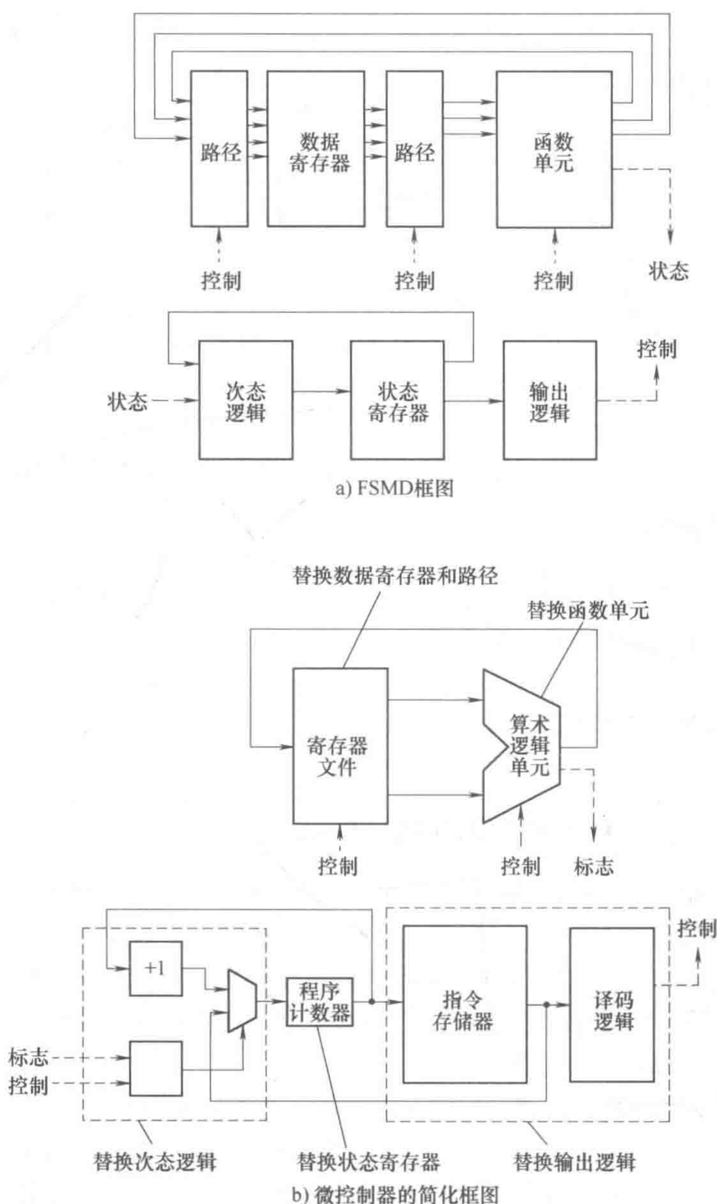


图 15-1　FSMD 和微控制器图

其次，我们可以使用可编程状态机来替换定制的 FSM，如图 15-1b 所示。运行 FSM 包含 3 个步骤：

- 状态机寄存器明确表示现态；
- 输出逻辑根据现态给出特定的输出信号；
- 次态逻辑决定新状态。

可编程状态机按照如下步骤对上述步骤进行修改：

- 使用程序计数器替代状态寄存器，计数器的内容表示控制路径上的现态；
- 在 FSM 中，每一个状态都产生确定的输出信号，从而来控制数据路径的操作，可编程状态机将这些输出模式编码成指令，并将其存储在一个存储器模块中，称为程序存储器或指令存储器，一个存储器地址对应一个程序计数器的状态（即一个数值），在执行过程中，需要从存储器中重新读取程序计数器所指向的指令，然后对其进行译码，从而得到控制信号，指令存储器和译码逻辑功能块是一个成熟的输出逻辑电路；
- 在 FSM 中，状态跳转到哪里是没有限制的，FSM 通过检查输入条件，可以从给定的现态跳转到多个可能的次态中的任一个，而在可编程状态机中，次态通常是现态加 1 的值（即程序计数器是逐 1 递增的），这恰好反映了顺序执行的本质，顺序执行只能被几个特定的指令所改变，例如 jump 指令，通过该指令程序计数器被装入不同的值，加法器及与其相关的多路逻辑功能块是一个简单的次态逻辑电路。

通过将数据路径替换为一个寄存器文件和一个 ALU，同时使用可编程状态机替换 FSM，定制系统的过程也就等同于开发一个指令次序（即开发一个软件程序）以及将指令加载到指令存储器中。FSMD 的组织架构对于不同的应用是不变的，形成了一个通用的硬件平台。该平台构成了 PicoBlaze 微控制器的基本骨架。

15.2.2 微控制器的应用

在定制的 FSMD 中，数据路径可以适应单一的应用需求。它可以包含多种定制的功能单元和并行路径，同时可以在单一状态（即一个时钟周期）中完成复杂的运算。另外，PicoBlaze 微控制器每次只能完成一个预先确定的 RT 操作（即一个指令）。若要完成相同的任务，则需要较多的指令，当然需要花费更多的时间。

许多任务可以通过一个定制 FSMD 或一个微控制器来完成。人们所寻求的就是硬件复杂度及性能与开发简易性之间的平衡。这里没有精确的准则来告诉人们去选择哪一个。因为开发软件的过程通常比制造定制硬件要简单，所以微控制器一般更适合对时序要求不是很严格的应用。我们可以通过分析计算的复杂度来确

定选择的可行性。PicoBlaze 执行一个指令需要两个时钟周期。假如系统时钟是 50MHz，25，000，000 条指令可以在 1s 内完成。对于一个任务（或一系列任务），我们可以分析需要多大的频率以及任务必须完成的速度，进而分析得到可用的指令的数目。例如，假定键盘接口每 1ms 产生新的输入数据，且数据必须在该时间段内处理。在 1ms 的周期内，PicoBlaze 能够执行 25，000 条指令。若某一必须的处理过程能够在 25，000 条指令内完成，PicoBlaze 控制器是一个可行的选择。总而言之，微控制器适用于许多对时序要求不严格的 I/O 接口或常规任务。

15.3 PicoBlaze 概述

15.3.1 基本组成

PicoBlaze 是一个紧凑的 8 位微控制器，其特征如下：

- 8 位数据宽度；
- 8 位 ALU（具有进位和零标志）；
- 16 个 8 位通用寄存器；
- 64 字节数据存储器；
- 18 位指令位宽；
- 10 位指令地址，支持最高 1024 条指令；
- 31 字调用/返回堆栈；
- 256 个输入端口和 256 个输出端口；
- 执行一条指令需两个时钟周期；
- 执行一个中断操作需 5 个时钟周期。

PicoBlaze 基于如图 15-1b 所示的框架，通过增加一些扩展可以使其功能更强大。其框图如图 15-2 所示。为了表示得更清晰，图中只给出了主要的数据流。主要存储部件的大小列在括号中。该处理器在初始框架的基础上可以做如下扩展：

- 增加 64 位字长的数据存储器，在 Xilinx 产品中被称为可擦除 RAM，在这里被称为数据 RAM，数据 RAM 可以被看做一个容器，用来存储额外的数据，需要注意的是，在数据 RAM 和 ALU 之间没有直接的通信路径，数据必须从寄存器中读出进行处理，然后再返存到数据 RAM 中；
- 在某些指令中增加立即数区，这使得常数，而非寄存器的内容，可以在 ALU 或其他操作中使用。在 ALU 底层输入前的 2-1 多路选择器被用来选择寄存器输出或常数区；

● 增加 31 字长的堆栈，支持功能调用，调用和返回流程将在 15.5.8 节详细描述；

● 增加路径实现外部数据的输入、输出，使用 8 位的 port_id 信号来确定端口，进而扩展到 256 个输入端口和 256 个输出端口，I/O 接口将在第 17 章详细描述；

● 增加中断处理电路（图中未给出），中断机制将在第 18 章详细描述。

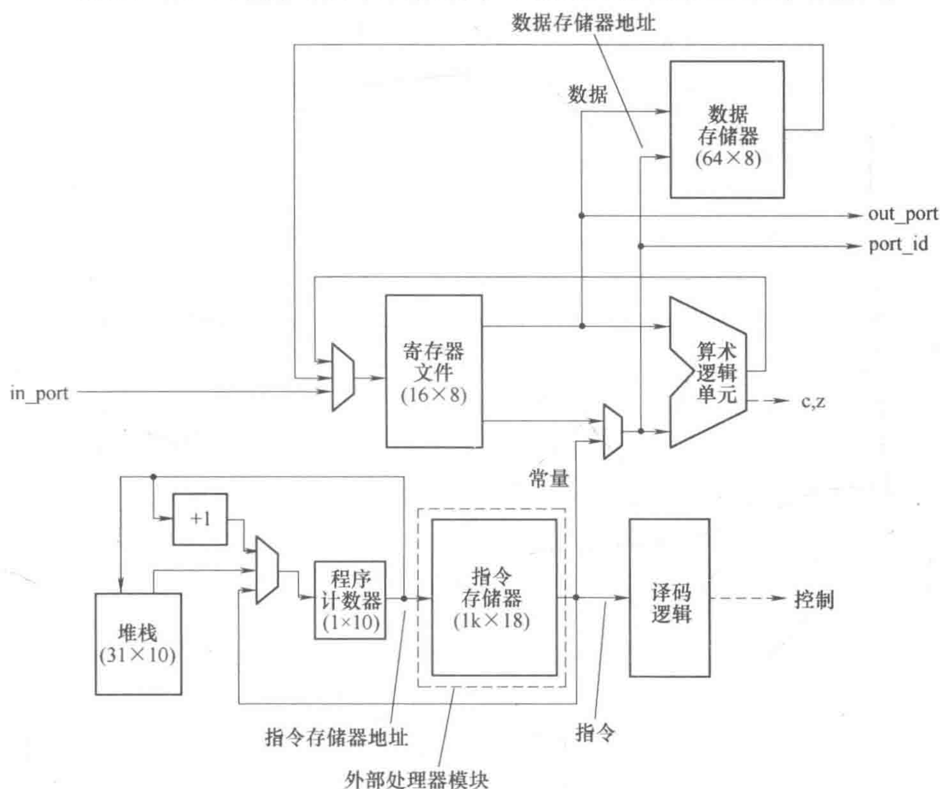


图 15-2　PicoBlaze 框图

15.3.2　顶层 HDL 模块

在综合过程中，PicoBlaze 系统由两个顶层模块构成，如图 15-3 所示。KCPSM3 模块是 PicoBlaze 处理器，表示由可编程状态机编码的常数（K），同时反映了 PicoBlaze 处理器的初始名称。其输入、输出信号如下所示：

● clk（输入，1 位）：系统时钟信号；

● reset（输入，1 位）：复位信号；

● address（输出，10 位）：指令存储器地址，指定读取指令时的位置；

● instruction（输入，18 位）：fetched 指令；

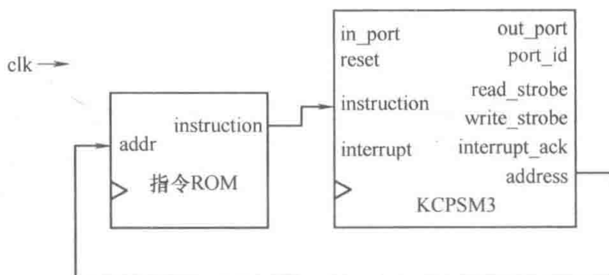


图 15-3　PicoBlaze 顶层图

- port_id（输出，8 位）：输入、输出端口地址；
- in_port（输入，8 位）：外部 I/O 输入数据；
- read_storbe（输出，1 位）：与输入操作相关的开关信号；
- out_port（输出，8 位）：输出给外部 I/O 数据；
- write_storbe（输出，1 位）：与输出操作相关的开关信号；
- interrupt（输入，1 位）：外部 I/O 中断请求信号；
- interrupt_ack（输出，1 位）：输出给外部 I/O 的中断应答信号。

第二个模块实现指令存储器。在开发过程中，通常将已编译的汇编代码预先存入存储器中，然后以 HDL 编码的形式将其配置成 ROM，即指令 ROM。

15.4　开发流程

在基于传统的微控制器开发系统时，需要分析需求的功能项，从而选择具有合适的计算能力以及 I/O 接口的处理器。通常还需要额外的芯片来实现特定的功能。使用软核微控制器的一大优势就是不仅可以使用其中的定制电路，而且还可以利用相同 FPGA 芯片中配置的微控制器。大量的应用通常包含许多不同的任务。在一个 FPGA 平台中，利用定制电路（即所谓的硬件）可以实现有严格时序要求的任务，而利用微控制器（即所谓的软件）则可以实现剩余的常规功能和低速 I/O 功能。

基于 PicoBlaze 的开发流程如图 15-4 所示，包含如下的步骤：

1）合理划分软件和硬件部分；

2）为软件部分开发汇编程序；

3）编译汇编程序，生成指令 ROM，该 ROM 是 HDL 文件；

4）执行指令集级别仿真；

5）获得硬件部分的 HDL 代码，该硬件包含用来实现特定的 I/O 功能和严格时序要求的电路功能以及具有 PicoBlaze 的接口；

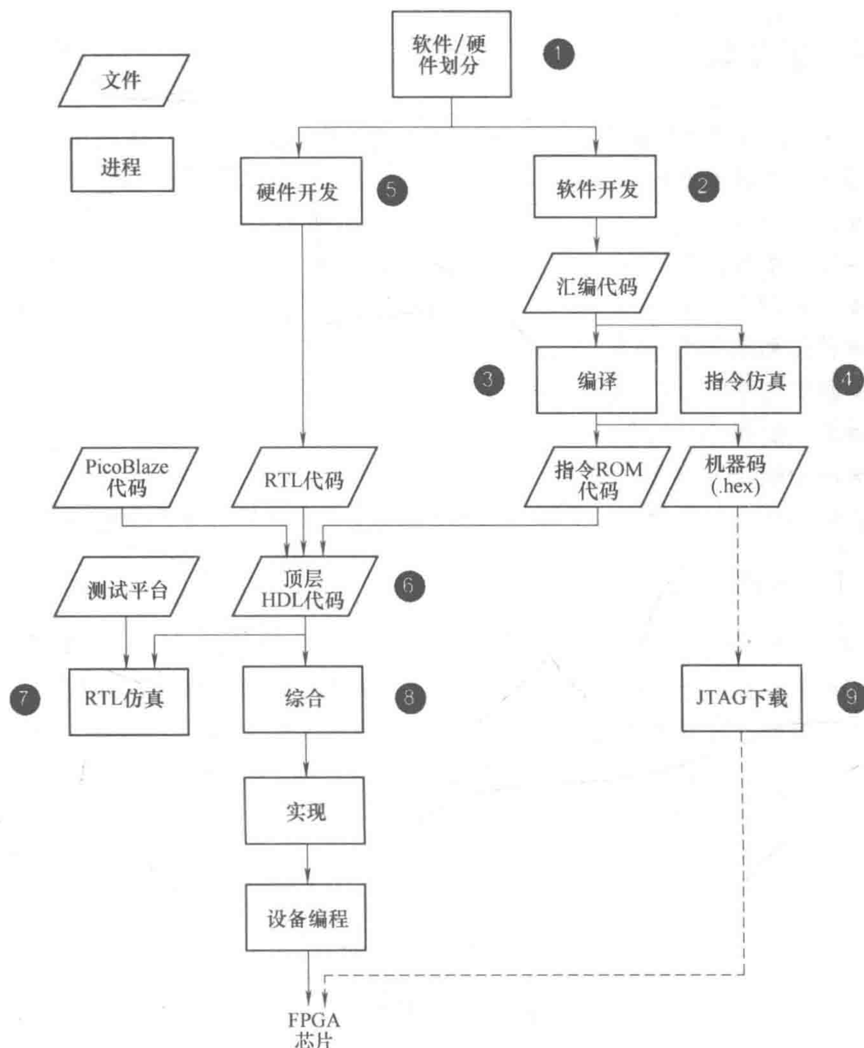


图 15-4 基于 PicoBlaze 的系统开发流程图

6）编制顶层 HDL 代码，实现与 PicoBlaze 核、指令 ROM 以及定制硬件的连接；

7）开发 testbench，进行整个系统的 HDL 级仿真；

8）完成 HDL 代码的综合实现，并在电路板上对 FPGA 芯片编程。

上述各步骤的详细描述将在后续章节中给出。

步骤 9 如图中虚线所示，该步骤不是正常开发流程的一部分。该步骤实现指令存储器在系统综合后的重加载。该步骤将在 16. 5. 3 节中讨论。

15.5 指令集

PicoBlaze 共有 57 条指令。这些指令具有 5 种格式。按照具体的执行形式将其分成如下几种类型：

- 逻辑指令；
- 算术指令；
- 比较和检验指令；
- 移位和循环指令；
- 数据传输指令；
- 编程流程控制指令；
- 与中断相关的指令。

在该部分，首先分析编程模式和指令格式，然后依次列出和解释各指令。

15.5.1 编程模式

以汇编编程为例，PicoBlaze 包含 16 个 8 位寄存器，1 个 64 字节的数据 RAM，3 个标志信号（零标志、进位标志和中断标志），编程计数器和堆栈栈顶指针。编程模式有时又被称作指令集架构，如图 15-5 所示。一条指令执行后，这些部分的内容会显式或者隐式地改变。与各指令相关的操作将在 15.5.3 节讨论。

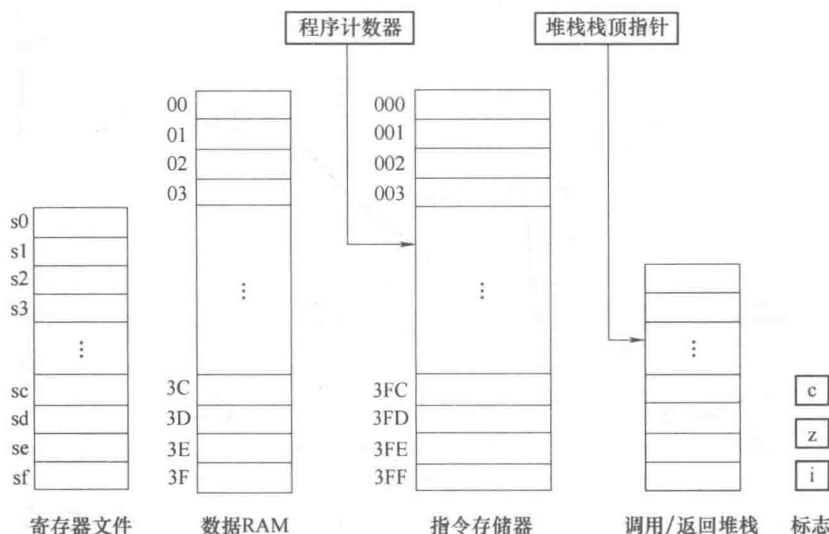


图 15-5 PicoBlaze 编程模式

使用如下所示的符号来表示存储器的组成部分以及一些常量定义：

- sX，sY：每一个代表 16 个通用寄存器中的其中之一，其中的 X 和 Y 表示 0 到 f 十六进制值；
- pc：程序计数器；
- tos：堆栈栈顶指针；
- c，z，i：进位标志、零标志和中断标志；
- KK：8 位常数或端口 id，通常被表示为两个十六进制数；
- SS：6 位常数，表示数据存储器地址，通常被表示为两个十六进制数；
- AAA：10 位常数，表示指令存储器地址，通常被表示为 3 个十六进制数。

15.5.2　指令格式

在汇编程序中，通常遵循 HDL 代码中的编程惯例：关键字（指令）用黑体描述，常量用大写字母描述。PicoBlaze 的指令有 5 种格式：

- **op** sX，sY：*register-register format*，op 表示操作，sX 和 sY 表示操作数，其中 sX 也被当做目的寄存器。即实现 sX←sX **op** sY 操作；
- **op** sX，KK：*register-constant format*，该格式与 register-register 格式类似，区别在于第二个操作数换成了立即数，即实现 sX←sX **op** KK 操作；
- **op** sX：*single-register format*，该格式用在移位和循环指令中，只包括一个操作数，即实现 sX←**op** sX 操作；
- **op** AAA：*single-address format*，该格式用在跳转和调用（call）指令中，AAA 表示指令存储器的地址。当指定的条件满足时，AAA 被装订到程序计数器中；
- **op**：*zero-operand format*，该格式用在一些多样的指令中，这些指令不包含任何操作数。

针对 PicoBlaze，有两种汇编程序：来自 Xilinx 的 KCPSM3 以及来自 Mediatronix 的 PBlazeIDE。这两种程序对于某些指令使用不同的记忆法。在接下来的章节中，PBlazeIDE 中使用的可选择的描述方式将在括号中注明。

15.5.3　逻辑指令

PicoBlaze 有 6 条逻辑指令，完成 and、or 和 xor 操作。指令完成两寄存器或寄存器与常数之间的位逻辑操作。进位标志 c 经常忽略不计。零标志 z 反映操作结果。指令的描述方式、简单介绍和伪操作如下所述：

- **and** sX，sY

  -位与操作

  -伪操作：

sX←sX & sY;

c←0;

- **and** sX, KK

  -位与操作

  -伪操作:

  sX←sX & KK;

  c←0;

- **or** sX, sY

  -位或操作

  -伪操作:

  sX←sX | sY;

  c←0;

- **or** sX, KK

  -位或操作

  -伪操作:

  sX←sX | KK;

  c←0;

- **xor** sX, sY

  -位异或操作

  -伪操作:

  sX←sX ^ sY;

  c←0;

- **xor** sX, KK

  -位异或操作

  -伪操作:

  sX←sX ^ KK;

  c←0;

15.5.4 算术指令

PicoBlaze 有 8 条算术指令，完成带有或不带有进位标志的加法和减法操作。进位标志 c 和零标志 z 反映操作结果。指令的描述方式、简单介绍和伪操作如下所述：

- **add** sX, sY

  -不带进位的加操作

  -伪操作:

sX←sX + sY;

- **add** sX, KK
  - -不带进位的加操作
  - -伪操作:
  - sX←sX + KK;
- **addcy** sX, sY(**addc** sX, sY)
  - -带进位的加操作
  - -伪操作:
  - sX←sX + sY + c;
- **addcy** sX, KK(**addc** sX, KK)
  - -带进位的加操作
  - -伪操作:
  - sX←sX + KK + c;
- **sub** sX, sY
  - -不带进位的减操作
  - -伪操作:
  - sX←sX-sY;
- **sub** sX, KK
  - -不带进位的减操作
  - -伪操作:
  - sX←sX-KK;
- **subcy** sX, sY(**subc** sX, sY)
  - -带进位的减操作(标志位被当做借位使用)
  - -伪操作:
  - sX←sX-sY-c;
- **subcy** sX, KK(**subc** sX, KK)
  - -带进位的减操作(标志位被当做借位使用)
  - -伪操作:
  - sX←sX-KK-c;

15.5.5　比较和检验指令

比较和检验指令对两寄存器或寄存器和常数进行检查分析，并相应地给出进位和零标志。寄存器的内容保持不变。该指令通常用在与条件性跳转指令或调用指令的连接上，其操作基于标志位的值。

比较指令执行减法操作，执行结果用来给出进位和零标志而不存入寄存器

中。指令的描述方式、简单介绍和伪操作如下所述：

- **compare** sX，sY(comp sX，sY)

  -对两寄存器进行比较并给出标志位

  -伪操作：

  if sX = = sY then z←1 else z←0;

  if sX > sY then c←1 else c←0;

- **compare** sX，KK(comp sX，KK)

  -对寄存器和常数进行比较并给出标志位

  -伪操作：

  if sX = = KK then z←1 else z←0;

  if sX > KK then c←1 else c←0;

检验指令执行与操作，结果用来给出标志位而不存入寄存器中。如果结果是0，则零标志置1。同时，结果输入到一个8输入的异或电路中，从而得到奇校验结果。若结果中包含奇数个1，则进位标志置1。指令的描述方式、简单介绍和伪操作如下所述。其中，t是8位的中间结果，在最终结果中被丢弃。

- **test** sX，sY

  -对两寄存器进行检验并给出标志位

  -伪操作：

  t←sX & sY;

  if t ==0 then z←1 else z←0;

  c←t[7] ^ t[6] ^ …^ t[0];

- **test** sX，KK

  -对寄存器和常数进行检验并给出标志位

  -伪操作：

  t←sX & KK;

  if t ==0 then z←1 else z←0;

  c←t[7] ^ t[6] ^ …^ t[0];

15.5.6　移位和循环指令

PicoBlaze 包含4个左移指令，4个右移指令和2个循环指令。这些指令按 single-register 格式执行且只有一个操作数，其图形化描述见图15-6。指令的描述方式、简单介绍和伪操作如下所述：

- **sl0** sX

  -左移1位，最低位补0

  -伪操作：

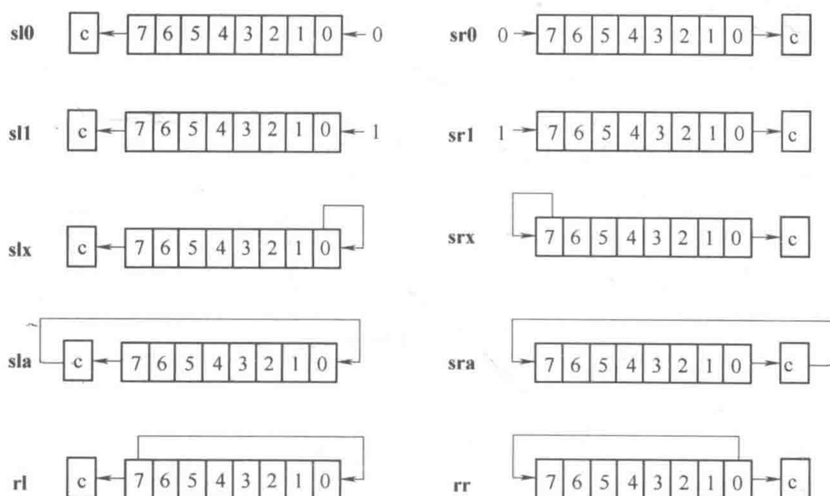


图 15-6　移位和循环指令图

sX←{sX[6：0]，0}；

c←sX[7]；

- **sl1** sX

  -左移 1 位，最低位补 1

  -伪操作：

  sX←{sX[6：0]，1}；

  c←sX[7]；

- **slx** sX

  -左移 1 位，最低位补 sX[0]

  -伪操作：

  sX←{sX[6：0]，sX[0]}；

  c←sX[7]；

- **sla** sX

  -左移 1 位，最低位补 c

  -伪操作：

  sX←{sX[6：0]，c}；

  c←sX[7]；

- **sr0** sX

  -右移 1 位，最高位补 0

  -伪操作：

  sX←{0，sX[7：1]}；

  c←sX[0]；

- **sr1** sX

  -右移 1 位，最高位补 1

  -伪操作：

  sX←{1, sX[7: 1]};

  c←sX[0];

- **srx** sX

  -右移 1 位，最高位补 sX[7]

  -伪操作：

  sX←{sX[7], sX[7: 1]};

  c←sX[0];

- **sra** sX

  -右移 1 位，最高位补 c

  -伪操作：

  sX←{c, sX[7: 1]};

  c←sX[0];

- **rl** sX

  -循环左移 1 位

  -伪操作：

  sX←{ sX[6: 0], sX[7]};

  c←sX[7];

- **rr** sX

  -循环右移 1 位

  -伪操作：

  sX←{ sX[0], sX[7: 1]};

  c←sX[0];

15.5.7　数据传输指令

PicoBlaze 中计算是通过寄存器和 ALU 来完成的。数据 RAM 提供附加的存储空间，而 I/O 端口提供与外设的通信路径。其中有些指令实现寄存器、数据 RAM 和 I/O 端口间的数据传输。数据传输指令可以划分为以下三种类型：

- 寄存器间：**load** 指令；
- 寄存器和数据 RAM 间：**fetch** 和 **store** 指令；
- 寄存器和 I/O 端口间：**input** 和 **output** 指令。

指令的描述方式、简单介绍和伪操作如下所述。其中，RAM [ ] 表示数据 RAM 的内容。需要指出的是，indirect address，如（sY）所示，被用来强调寄存

器 sY 中的内容已被使用。

- **load** sX, sY
  - -在两寄存器间进行数据传输
  - -伪操作：
    - sX←sY
- **load** sX, KK
  - -将常数输入寄存器
  - -伪操作：
    - sX←KK
- **fetch** sX, (sY)(**fetch** sX, sY)
  - -将数据从数据 RAM 输入到寄存器
  - -伪操作：
    - sX←RAM[(sY)];
- **fetch** sX, SS
  - -将数据从数据 RAM 输入到寄存器
  - -伪操作：
    - sX←RAM[SS];
- **store** sX, (sY)(**store** sX, sY)
  - -将数据从寄存器输入到数据 RAM
  - -伪操作：
  - RAM[(sY)] ← sX;
- **store** sX, SS
  - -将数据从寄存器输入到数据 RAM
  - -伪操作：
    - RAM[SS] ← sX;
- **input** sX, (sY)(**in** sX, sY)
  - -将数据从输入端口传输到寄存器
  - -伪操作：
    - port_id ← sY;
    - sX←in_port;
- **input** sX, KK(**in** sX, KK)
  - -将数据从输入端口传输到寄存器
  - -伪操作：
    - port_id ← KK;
    - sX←in_port;

- **output** sX，(sY)(**out** sX，sY)

    -将数据从寄存器传输到输出端口

    -伪操作：

        port_id ← sY;

        out_port ←sX;

- **output** sX，KK(**out** sX，KK)

    -将数据从寄存器传输到输出端口

    -伪操作：

        port_id ← KK;

        out_port ←sX;

PicoBlaze 中没有指令可以直接实现将数据传输到指令存储器或者将数据从指令存储器中取出。然而，许多指令包含立即数存储区。因为常数是指令的一部分且存储在指令存储器中，因此可以认为暗含将数据从指令存储器传输到寄存器中。

15.5.8 程序控制指令

在 PicoBlaze 中，程序计数器用以标识指令存放的地址。默认情况下，进程执行到指令寄存器的下一个地址时程序计数器自动加一（即：pc←pc + 1）。jump、call 和 return 指令会载入某一固定值用于程序计数和程序控制。这些指令会被无条件执行或依据 carry 和 zero 标志的值选择性执行。

如果相应的条件满足，jump 指令会向程序指针载入一个新的变量值。程序将跳出当前的操作跳转到新的地址空间中。该指令执行结束后程序继续先前的操作。指令的关键字、相关描述以及示例操作如下所示。重申一下：AAA 表示 10bit 的指令寄存器地址，PC 表示程序计数器。

- Jump AAA

    ---无条件跳转

    --- 操作示例：

      Pc←AAA

- Jump c，AAA

    ---carry 置一时程序跳转

    ---操作示例：

      If c == 1 then pc←AAA else pc←pc + 1;

- Jump nc，AAA

    ---carry 置零时程序跳转

    ---操作示例：

      If c == 0 then pc←AAA else pc←pc + 1;

● Jump z，AAA

---zero 置一时程序跳转

---操作示例：

If z == 1 then pc←AAA else pc←pc + 1；

● Jump nz，AAA

---zero 置零时程序跳转

---操作示例：

If z == 0 then pc←AAA else pc←pc + 1；

Call 和 return 指令用于软件函数的调用。当函数被调用时，处理器中止当前的操作并跳转到相应的程序中。当该程序执行结束后，处理器将返回到跳转点继续执行先前的操作。和 jump 指令一样，如果条件合适的话 call 命令也是向程序计数器载入一个新的变量值。另外，它也是将当前程序计数器的值存储在一个特殊的缓存器中，该缓存器叫做堆栈。新的地址表示程序执行的起始节点。程序执行结束后需要包含一个 return 指令。return 指令从堆栈中获取保存好的变量值后自动加一，载入到程序计数器中。该指令可以让操作立即返回到原 call 指令调用的地址空间中。典型的程序流程见图 15-7。

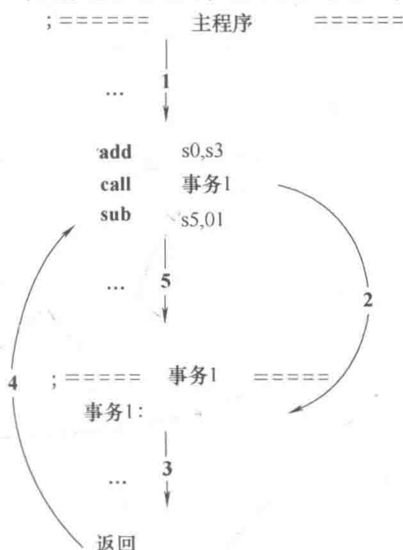


图 15-7 子函数调用的程序执行过程

PicoBlaze 支持函数嵌套调用，这意味着一个函数可以被其他的函数调用。为了实现这个功能，我们用一个后进先出的栈来存储程序计数器的值。在这个缓冲器中，最新调用的函数地址被压到栈顶（即：后进）。假设这个函数里面不再调用其他的函数。它优先被执行完成后将返回的地址保存到栈顶。该地址从堆栈中弹出（即：先出）后恢复到先前的操作。PicoBlaze 提供了 31 字节的堆栈用于嵌套函数的调用和返回操作。

call、return 指令的关键字、相关描述以及示例操作如下所示。重申一下：tos 信号用于表示栈顶指针。STACK [ ] 表示堆栈的内容。

● Call AAA

---无条件调用子函数

---操作示例：

Tos←tos + 1；

STACK [tos] ←pc;
Pc←AAA

- Call c, AAA

---carry 置一时调用子函数

---操作示例:

If c == 1 then
Tos←tos + 1;
STACK [tos] ←pc;
Pc←AAA;
Else
Pc←pc + 1;

- Call nc, AAA

---carry 置零时调用子函数

---操作示例:

If c == 0 then
Tos←tos + 1;
STACK [tos] ←pc;
Pc←AAA;
Else
Pc←pc + 1;

- Call z, AAA

---zero 置一时调用子函数

---操作示例:

If z == 1 then
Tos←tos + 1;
STACK [tos] ←pc;
Pc←AAA;
Else
Pc←pc + 1;

- Call nz, AAA

---zero 置零时调用子函数

---操作示例:

If z == 0 then
Tos←tos + 1;
STACK [tos] ←pc;

```
    Pc←AAA;
  Else
    Pc←pc + 1;
```

- Return（ret）

  ---无条件返回

  ---操作示例:

```
  pc←STACK [tos] +1;
  tos←tos - 1;
```

- Return c（ret c）

  ---置一时返回

  ---操作示例:

```
  If c ==1 then
    pc←STACK [tos] +1;
    tos←tos - 1;
  Else
    Pc←pc + 1;
```

- Return nc（ret nc）

  ---carry 置零时返回

  ---操作示例:

```
  If c ==0 then
    pc←STACK [tos] +1;
    tos←tos - 1;
  Else
    Pc←pc + 1;
```

- Return z（ret z）

  ---zero 置一时返回

  ---操作示例:

```
  If z ==1 then
    pc←STACK [tos] +1;
    tos←tos - 1;
  Else
    Pc←pc + 1;
```

- Return nz（ret nz）

  ---zero 置零时返回

  ---操作示例:

```
If nz ==0 then
  pc←STACK [tos] +1;
  tos←tos -1;
Else
  Pc←pc +1;
```

15.5.9 中断指令

中断是另一种改变程序执行过程的机制，这将会在第 18 章进行一个详细的阐述。不同于 jump 和 call 指令，它需要一个外部的请求来触发。当中断标志被激活并且中断请求被声明时，PicoBlaze 开始实施当前的命令，在声明/返回的栈中保存下一条指令的地址，保存 carry 和 zero 标志，屏蔽中断标识，同时将程序计数器的值设为中断服务程序的起始地址 3FF。PicoBlaze 有两个中断返回指令，用于从中断位置中恢复操作。还有两个指令用于通过设定或清除中断标志来使能或者屏蔽中断请求。这些指令的关键字、相关描述以及示例操作如下：

- Returni disable(reti disable)

  ---从中断服务程序中返回同时屏蔽中断标志位

  ---示例操作：

  ```
  Pc←STACK[tos];
  Tos←tos - 1;
  I←0;
  C←preserved c;
  Z←preserved z;
  ```

- Returni enable (reti enable)

  ---从中断服务程序中返回同时使能中断标志位

  ---示例操作：

  ```
  Pc←STACK[tos];
  Tos←tos - 1;
  I←1;
  C←preserved c;
  Z←preserved z;
  ```

- Enable interrupt(eint)

  ---使能中断请求

  ---示例操作：

  ```
  I←1;
  ```

- disable interrupt(dint)

---屏蔽中断请求

---示例操作:

I←0;

注意：中断程序中保存了下一条指令的地址。当 returni 指令被执行，存储在栈顶(即：STACK[tos])的操作地址将会被恢复。这与一般的 return 指令不同，通常的 return 指令是将存储的地址加一后恢复用于操作(及 STACK[tos]+1)。

15.6 伪指令声明指令

伪指令像是一个汇编程序指令。虽然，它不是微处理器指令设置的一部分，但其经常有助于程序开发。像它名字所描述的那样，一个伪指令表示汇编程序完成一个特定的任务，像定义一个常量或者保留存储空间。KCPSM3 汇编和 PBlazeIDE 汇编在伪指令方面稍微有一些不同，具体内容将在下面的模块中给出。

15.6.1 KCPSM3 汇编伪指令

在 KCPSM3 汇编中主要伪指令的关键字、相关描述和示例操作如下：

- Address

  ---这条伪指令是将特定的程序存储到特定的 ROM 地址中。

  ---例如:

  Address 3FF

- Namereg

  ---这条伪指令是将一个寄存器赋予一个符号名。这将便于程序更好的描述。

  ---例如:

  Namereg s5, index

- Constant

  ---这条伪指令将一个常量赋予符号名。这将便于程序更好的描述。

  ---例如:

  Constant max, F0

15.6.2 PBlazeIDE 汇编伪指令

在 PBlazeIDE 汇编中主要伪指令的关键字、相关描述和示例操作如下。注意：$符号表示该数据用十六进制表示。

- Org

  ---这条伪指令详细说明了后面的程序代码存储到指令 ROM 的特定的

地址空间中(即，“originate”存储在此地址空间中)。

---例如：

Org $ 3FF

- Equu

---这条伪指令为一个变量或寄存器设置一个“等同”的符号。它为一个常量或寄存器设定一个符号名。

---例如：

```
Max     equ   128/8
Index   equ   S5
```

- Dsin，dsout，dsio

---这些伪指令为 I/O 端口的 id 设定符号名。相应的端口可以设定为输入、输出或者双端口信号。这些指令和 equ 之间的区别在于 PBlazeIDE 生成一个“端口指示器”用于在仿真界面上显示这些伪指令。在仿真过程中这些端口的动态将会通过指示器显示出来。

---例如：

```
Keyboard  dsin $ OE
Switch    dsin $ OF
Led       dsout $ 15
```

- Vhdl

---这条伪指令用于生成一个 VHDL 类型的 ROM。具体内容将会在第 16 章详细说明。

---例如：

Vhdl “template. vhd”，“target. vhd”，“ROM”

15.7 文献备注

PicoBlaze 手册来自于 xilinx 公司的“PicoBlaze 8-bit 嵌入式微处理器用户指南”，它提供了详细的微处理器信息，包括硬件组成、指令设置、开发流程以及 KCPSM3 和 PBlazeIDE 汇编语言的描述。PicoBlaze 公司的设计师 Ken Chapman 在“创造嵌入式微处理器”一文中描述了此微处理器的来源，这在 Xilinx 网站的 TechXclusives 中是非常的有用。

KCPSM3 汇编、PicoBlaze HDL 语言和指令 ROM 的 HDL 模型可以在 Xilinx 网站中下载。搜索关键字“PicoBlaze”可以直接跳转到下载界面。PBlazeIDE 汇编语言可以到 Mediatronix 的官方网站 http：//www. mediatronix. com 下载。该网站还提供了更多的关于该软件的详细信息。

第16章　PicoBlaze 汇编语言开发

16.1　简介

因为其简易性，PicoBlaze 不能有效地支持高级程序语言，所以它一般使用汇编语言开发。在这一章，我们提供了一个由顶向下的完整的开发流程。首先我们介绍数据处理速率和程序控制方法，其次是关于子程序的使用，最后引出整个系统结构的要点。

16.2　有效的代码段

PicoBlaze 微处理器包含了字节类型的数据处理和简单的条件分支指令。在这一部分，我们将举例说明怎样构造代码去实现单比特和多字节操作，同时了解怎样频繁调用高级语言去实现控制。

16.2.1　KCPSM3 协议

KCPSM3 汇编语言在设计中使用如下协议：

- 程序中在地址符后使用一个“:”，如“code:”；
- 在注释之前使用“;”；
- 为一个常数加 HH，表示该数据为十六进制数据。

一段示例性的代码如下：

```
; 这是一个演示程序段
    test S0, 82; 将S0 和10000010 进行比较
    jump z, clr-sl; 如果S0 的最低位为0，程序跳转到dr-sf
    load sl, FF; 否则，将1111-1111 赋值给S1
    clr-sl:
    load sl, 01; 将00000001 赋给S1
```

16.2.2　比特操作

PicoBlaze 的指令设置主要用于字节型操作。比特型操作经常被用于控制底层的 I/O 端口操作，比如测试、设置和清除 1bit 标志信号。

要操作一个单比特信号，我们首先要定义一个掩码（即 mask）来隔离和保存无关比特信号，然后在特定的比特上执行特定的操作（即 unmasked）。我们可以通过与合适掩码执行 or、and 和 xor 操作来设定、清除和转换（即转化）数据的一些比特位。下面的程序段表示怎样去设定、清除和转换寄存器 s0 的第二个低比特位。

```
constant SET-MASK, 02; mask =0000_ 0010
constant CLR-MASK, FD; mask =1111 -1101
constant TOG-MASK, 02; mask =0000-0010
or      S0, SET-MASK; 设置第二个低比特位为1
and     S0, CLR-MASK; 清除第二个低比特位
xor     S0, TOG-MASK; 转换第二个低比特位
```

以上掩码隔离操作基于如下运算，对于所有的布尔型变量 x，$x\oplus 0=x$ 、$x\oplus 1=x'$。同样的法则还可以应用于多比特数据转换。例如，我们可以通过如下操作清除高位数据（即，4 个高位）

```
and s0,0F              ;mask =0000 _1111
```

我们也可以将掩码的概念使用到测试指令中，用来检测其中的某一位。例如，下面程序段根据测试寄存器 s0 的高位数据来确定程序跳转的分支：

```
test S0, 80;            mask =1000 _0000
jump nz, msb-set;       如果高位为1，则跳转到msb-set 分支
; 高位不为1 时程序执行分支
jump done
msb-set:
; 高位为1 时程序执行分支
…
done :
…
```

一个单独的比特信号可以通过前面的代码提取出来。例如，下面的程序段用于提取寄存器 s0 的高位数据并将其存储到寄存器 s1 中：

```
Load    s1, 00
test    S0, 80          ; mask =1000-0000，提取高位
jump    z, done         ; 正确，低位置零
load    sl, 01          ; 错误，将01 赋给S1
done:
…
```

16.2.3 多字节数据处理

一个微处理器有时候需要处理大量的多字节数据，比如一个大的计数器。当 PicoBlaze 的数据位宽为 8bit 时，处理此类数据需要一个在两个连续指令中传递信息的机制。PicoBlaze 使用进位标志来达到这个目的。对于算术指令，它们将加和减译为一个需要进位一个不需要进位，就像 add 和 addcy 命令一样。对于移动和翻转指令，进位被移到一个寄存器的高位或者低位，反之亦然。

设 x 和 y 均为 24bit 的数据，故它们都需要占用 3 个寄存器。下面的程序段说明了多字节数据相加过程中进位信号的使用方法：

```
namereg s0, x0; x 的最低位字节
namereg sl, xl; x 的中间字节
namereg s2, x2; x 的最高位字节
namereg s3, y0; y 的最低位字节
namereg s4, yl; y 的中间字节
namereg s5, y2; y 的最高位字节
; add: (x2, xl, x0) + (y2, yl, y0)
add   x0, y0; 最低位字节相加
addcy xl, yl; 带有进位的中间字节相加
addcy x2, y2; 带有进位的最高位字节相加
```

第一条指令实现最低位字节相加并将进位值存储到进位标志中。第二条指令将包含有进位标志的中间字节相加。同样的，第三条指令使用先前相加所得到的带有进位标志的结果与最高位字节相加。

多字节的加减法可以通过类似的方式实现：

```
; increment: (x2, xl, x0) +1
add   x0, 01; 最低位字节累加
addcy xl, 00; 带进位的中间字节累加
addcy x2, 00; 带进位的最高字节累加
; subtract: (x2, xl, x0) - (y2, yl, y0)
sub   x0, y0; 最低位字节相减
subcy xl, yl; 带借位的中间字节相减
subcy x2, y2; 带借位的最高字节相减
```

多字节数据转换可以通过包含进位标志的单独的移位运算指令实现。例如，sla 指令是将数据左移一位并将进位移动到该数据的最低位。实现 3 字节数据左移的程序如下所示：

```
; shift (x2, xl, x0) via carry
```

s10 x0；*将x0 的低位补零，最高位移到进位中*
sla x1；*将x1 的进位移到最低位，最高位移到进位中*
sla x2；*将x2 的进位移到最低位，最高位移到进位中*

16.2.4 控制结构

一种高级语言通常包含各种各样的控制结构去改变程序执行的顺序。这包括 if-else 语句、case 语句和 for 循环语句。另一方面，PicoBlaze 只提供有条件和无条件调转指令。尽管指令简单，我们可以通过它们和 test 或 compare 指令一起去实现高层次的控制结构。下面的例子说明了 if-else 语句、case 语句和 for 循环语句的构造。

首先我们介绍 if-else 语句：

```
if( s0 == sl) {
    /*then 分支状态*/
}
else{
    /*else 分支状态*/
}
```

相应的汇编代码段如下：

```
    compare s0, sl
    jump nz, else-branch
; then 分支的代码
    ...
    jump if-done
else-branch:
    ; else 分支的代码
    ...
if-done:
; 如果状态有效执行下面代码
    ...
```

程序中使用 compare 指令去检查条件 s0 == s1 和设定 zero 标志。下面的 jump 指令检查相应的判定标志是否为 0 来确定程序是否跳转到 else 分支。

Case 结构可以看作是一个多路跳转，这个结构的跳转取决于条件表达式的值。下面的程序中将变量 s0 作为表达式并跳转到相应的分支：

```
switch (s0){
    case value1:
```

```
        /*value1 状态时程序执行的操作*/
        break;
    case value2:
        /*value2 状态时程序执行的操作*/
    break;
    case value3:
        /*value3 状态时程序执行的操作*/
        break;
    default:
        /*default 状态时程序执行的操作*/
}
```

一些处理器的多路跳转可以通过一种"地址索引模式"的硬件特性实现。然而，PicoBlaze 不支持这种特性，所以 case 指令不得不被看作一个 if-else 序列。换句话说，前面的 case 语句可以看作如下操作：

```
if ( s0 == value1 ) {
/*value1 状态时程序执行的操作*/
}
elseif( s0 == value2) {
/*value2 状态时程序执行的操作*/
}
elseif( s0 == value3) {
/*value3 状态时程序执行的操作*/
}
else(
/*default 状态时程序执行的操作*/
}
```

相应的汇编语言代码段变为

```
constant value1,
constant value2,
constant value3,
compare S0, value1; 测试value1
jump nz, case _2; 与value1 值不相等，则跳转
; case1 状态的代码
jump case-done
case _2:
```

```
compare S0, value2; 测试value2
jump nz, case _3; 与value2 值不相等，则跳转
; case2 状态的代码
jump case _ done
case _ 3:
compare S0, value3; 测试value3
jump default; 与value3 值不相等，则跳转
; case3 状态的代码
...
jump case _ done
default:
; default 的代码
...
case _ done :
; case 语句下面的代码
...
```

For 循环语句是重复执行一段程序代码。程序通过计数器的值来追踪循环执行的次数。具体示例如下所示：

```
For(I = MAX, I =0, I-1){
        /* 循环体 */
    }
```

汇编语言程序段如下：

```
    namereg s0, i                  ; 循环索引值
    constant MAX, . . .            ; 循环边界值
    load i, MAX; 载入循环索引值
loop-body:
; 循环体代码
    ...
    sub i, 01                      ; 循环索引值自减?
    jump nz, loop-body; done?
    : 下面是循环执行的代码
    ...
```

16.3　子程序开发

一个子程序，像C语言中的一个函数，实现一个庞大设计的一部分功能。它用于实现特定的功能并且可以重复使用。使用子程序允许我们方便地将一个设计分为若干个小的模块，这样既便于项目管理又可以提高设计的可靠性和可读性。这是一种基于先进设计理念的设计方法，同时它支持高级语言的开发。

PicoBlaze使用call和return指令去实现子程序调用。Call指令保存当前程序计数器的内容并将进程跳转到子程序的起始地址中。一个子程序通过return指令结束，它用于恢复保存在程序计数器的值并继续先前的操作。一个典型的流程图见图15-7。说明：PicoBlaze只有在函数调用和返回的时候才会保存和恢复程序计数器的值。我们不得不手动管理寄存器和数据RAM来保证在子函数执行完成后原系统中这些数据不会发生改变。

下面这个乘法的例子说明了一个子程序的开发过程。我们设输入为两个8bit的无符号整形数据，输出为一个16bit数据。运算法则基于简单的shift-and-add方法。这种方法是反复通过一个8bit的乘法器。在每次循环中，被乘数左移一个位置。如果相应的乘数为1，将转换后的被乘数加到运算结果中。汇编程序见示例16.1。被乘数和乘数分别被存储到寄存器s3和s4中。单比特的乘法通过重复右移s4得到，它将进位值移到数据的最低位。说明：和左移被乘数不同，我们将分别存储于寄存器s5和s6的由两个字节组成的运算结果右移。

示例16.1　软件整数乘法

```
=====================================================================
  程序：乘法软件
  作用：8bit 无符号数乘法使用shift-and-add 运算法则
  输入寄存器：
  s3：被乘数
  s4：乘数
  输出寄存器：
  s5：输出结果的高字节
  s6：输出结果的低字节
  临时寄存器：i
=====================================================================
mult-sof t：
      load s5，00                 ；清除s5
```

```
    load i, 08              ; 初始化循环索引值
mult-loop:
    sr0 s4                  ; 将最低位移到进位中
    jump nc, shift-prod     ; 最低位设置为0
    add s5, s3              ; 最低位设置为1
shift-prod:
    sras5                   ; 右移高位字节
                            ; 进位移到最高位，最低位移到进位中
    sras6                   ; 右移低字节数据
                            ; s5 的低位赋给s6 的高位
    sub i, 01               ; 循环索引值自减
    jump nz, mult-loop      ; 反复执行直至i=0
    return
```

由于汇编语言本身的特性，汇编语言的编程描述十分接近于底层元器件。一个子程序需要包含一个描述性的头文件和详细的注释。典型的头文件在示例16.1 具体说明。它由一个简短的功能描述和有效的寄存器组组成。后面将会介绍在一个大的设计中寄存器如何分配以及如何避免关键寄存器冲突。

16.4 编程

汇编语言程序的编写步骤如下：

1）编写主程序的伪代码；

2）分析主程序中包含的各任务，提取出来作为子程序。如果有必要，可将复杂子程序进一步分解；

3）确定寄存器和 RAM 的使用；

4）编写各子程序的源代码。

步骤 1、2 和 4 主要采用了“化整为零，各个击破”的方法，该方法适用于任何软件程序的编写。微控制器一般应用于简单的嵌入式系统中，在该系统中处理器不断地监控 I/O 接口并对其做出响应。它的主程序一般有如下结构：

```
    call initialization_routine
forever:
    call taskl_routine
    call task2_routine
    ...
```

```
    call taskn_routine
    jump forever
```

步骤 3）是汇编语言编程区别于高级语言而特有的。用高级语言编程时，编译器会自动地为变量分配存储单元，而用汇编语言编程时，我们必须人工分配好数据存储单元。PicoBlaze 有 16 位的寄存器和 64 字节的数据 RAM 用来存储数据。寄存器可看作快速存储单元，能直接对其内部数据进行操作。与之相对，数据 RAM 是“辅助”存储单元，其内数据需要转移到寄存器才能被处理。例如，如果我们想为 RAM 中的某一数据增加一个条目，必须首先将其移到一个寄存器中，增加完成之后再移回到 RAM 中。

数据 RAM 的使用在数据存储速率方面受到限制，这是需要提前考虑的因素，尤其是在代码较复杂且包含嵌套子程序的情况下。为了有助于编码，我们可以首先定义所需的全局存储单元或局部存储单元。前者存储整个程序用到的数据，后者为中间结果提供存储空间，相关运算结束后这部分空间得到释放。

16.4.1　示例

这个过程用一个例子来说明。让我们考虑一个程序，它用到前面的乘法子程序。它首先从转换器读入两个数据 a 和 b，然后计算 a^2+b^2，并将结果显示在 8 个离散的 LED 上。因为 I/O 接口将在第 17 章讲解，现我们仅将 8bit 的转换器看作一个输入端，将 8bit LED 看作输出端。我们假设 a 和 b 分别从转换器的高 4 位和低 4 位获取。主程序如下：

```
    call clear-data-ram
forever :
    call read-switch
    call square
    call write-led
    jump forever
```

子程序定义如下：

- clr_data_mem：系统初始化时清除数据存储器；
- read-switch：获取转换器高 4 位和低 4 位数据，将值存入数据 RAM；
- square：运用乘积子程序计算 a^2+b^2；
- write-led：将运算结果的低 8 位写入到 LED 端。

为了实现该示例的功能，在 read-switch 子程序中我们设计两个小程序 get_upper_nibble 和 get_lower_nibble 来获取数据高 4 位和数据低 4 位。

实现过程的第二步是分配寄存器和数据 RAM。我们为全局变量存储单元引入一个全局寄存器 sw_in 来存储从转换器读入的数据，并且分配 11 字节的数据

RAM 存储 square 子程序的输入和结果数据。数据 RAM 的分配如图 16-1 所示。注意地址 01 和 03 实际上并未使用。它们被保留用来简化 7 段式 LED 显示代码（相关内容将在第 17 章进行详细讲解）。所有剩余的寄存器被用作局部存储器。为使编程更清晰，我们定义 3 个信号名称 data、addr 和 i 作为 data、port and memory address 和 loop index 的临时寄存器。

地址	内容
00	低四位a
01	未使用
02	低四位b
03	未使用
04	低四位a^2
05	高四位a^2
06	低四位b^2
07	高四位b^2
08	低四位a^2+b^2
09	高四位a^2+b^2
0A	进位a^2+b^2

图 16-1　数据 RAM 存储器分配

最后一步是实现子程序的汇编语言编码。全部的源代码见示例 16.2。clr_data_mem 子程序用一个循环控制来清空数据存储区。寄存器 i 内是循环指针，初始化为 64（40H）。循环指针依循环次数递减，且 0 分配给了相应的数据 RAM 地址。write_led 程序从数据 RAM 中读取计算结果的低 8 位并输出给 LED 端。

子程序 read-switch 包含两个小程序，get_upper_nibble 子程序用来将 data 寄存器内的值右移 4 位以将高 4 位移至低 4 位。get_lower_nibble 子程序将高 4 位清为 0 用以移走高位数据。read-switch 程序的“glue instructions”用来输入转换器数值，提供高 4 位和低 4 位数据处理子程序的输入，并且将结果存入数据 RAM。

子程序 square 从数据 RAM 中取走数据，利用 mult_soft 子程序计算 a^2 和 b^2，求和，并且将计算结果存回数据 RAM。

示例 16.2　具有简单高四位和低四位输入的平方程序

```
; ---------------------------------------------------------------------------
; 具有简单I/O 接口的平方电路
; ---------------------------------------------------------------------------
; 程序操作：
;- a （高4 位）和b （低4 位）的读转换器
;- 计算a *a +b *b
;- 在8 个LED 上显示数据
; ---------------------------------------------------------------------------
; 数据常量
; ---------------------------------------------------------------------------
constant   UP_NIBBLE_MASK, OF ; 00001111
; ---------------------------------------------------------------------------
; 数据RAM 地址别名
; ---------------------------------------------------------------------------
```

```
constant  a_lsb, 00
constant  b_lsb, 02
constant  aa_lsb, 04
constant  aa_msb, 05
constant  bb_lsb, 06
constant  bb_msb, 07
constant  aabb_lsb, 08
constant  aabb_msb, 09
constant  aabb_cout, 0A
; ---------------------------------------------------------------------------------
; 寄存器别名
; ---------------------------------------------------------------------------------
; 通用的局部变量
namereg  s0, data ; 临时数据寄存器
namereg  sl , addr ; 临时储存器与I/O 端口地址寄存器
namereg  s2, i      ; 通用的循环指针
; 全局变量
namereg sf, sw_in
; ---------------------------------------------------------------------------------
; 端口别名
; ---------------------------------------------------------------------------------
; -------------------输入端口定义-------------------------
constant  sw_port, 01 ;8-bit  switches
; -------------------输出端口定义-------------------------
constant  led_port, 05
; ---------------------------------------------------------------------------------
; 主程序
; ---------------------------------------------------------------------------------
; 调用层次:
;
; main
; -clr_data_mem
; -read_switch
;       -get_trpper_nibble
;       -get_lower_nibble
```

```
;    -square
;       -mult_soft
;    -write_led
;
  call   clr_data_mem
forever:
  call   read_switch
  call   square
  call   write_led
  jump   forever
; ------------------------------------------------------------------------------------
; 程序：clr_data_mem
;   功能：清除数据存储器
;   临时寄存器：data , i
; ------------------------------------------------------------------------------------
  clr_data_mem:
           load   i, 40                    ; 初始化循环指针为64
           load   data, 00
  clr_mem_loop:
           store   data, (i)
           sub   i, 01                     ; 循环指针递减
           jump nz, clr_mem_loop           ; 重复执行直到i = 0
           return
; ------------------------------------------------------------------------------------
; 程序：read_switch
;   功能：从输入得到高四位和低四位
;   输入寄存器：sw_in
;   临时寄存器：data
; ------------------------------------------------------------------------------------
  read_switch:
         input   sw_in, sw_port            ; 读转换器输入
         load   data, sw_in
         call   get_lower_nibble
         store   data, a_lsb               ; 将a 存入数据RAM 中
         load   data, sw_in
```

```
        call   get_upper_nibble
        store  data, b_lsb                ; 将b 存入数据RAM 中
; ------------------------------------------------------------------------
; 程序: get_lower_nibble
;    功能: 得到数据低四位
;    输入寄存器: data
;    输出寄存器: data
; ------------------------------------------------------------------------
get_lower_nibble:
        and    data, UP_NIBBLE_MASK      ; 将高四位清零
        return
; ------------------------------------------------------------------------
; 程序: get_upper_nibble
;    功能: 得到数据高四位
;    输入寄存器: data
;    输出寄存器: data
; ------------------------------------------------------------------------
get_upper_nibble:
        sr0    data          ; 向右移位4 次
        sr0    data
        sr0    data
        sr0    data
        return
; ------------------------------------------------------------------------
; 程序: write-led
;    功能: 将结果的低八位输出给8 个LED
;    临时寄存器: data
; ------------------------------------------------------------------------
write-led:
        fetch  data, aabb_lsb
        output data, led_port
        return
; ------------------------------------------------------------------------
; 程序: square
;    功能: 计算a *a + b *b
```

```
;        将数据/结果存储在RAM 中
;   临时寄存器：s3 , s4 , s5 , s6 , data
; --------------------------------------------------------------------------------
square:
        ; 计算a *a
        fetch   s3, a_lsb           ; 加载a
        fetch   s4, a_lsb           ; 加载a
        call     mult_soft          ; 计算a *a
        store   s6, aa_lsb          ; 存储a *a 的低字节
        store   s5, aa_msb          ; 存储a *a 的高字节
        ; 计算b *b
        fetch   s3, b_lsb           ; 加载b
        fetch   s4, b_lsb           ; 加载b
        call   mult_soft            ; 计算b *b
        store   s6, bb_lsb          ; 存储b *b 的低字节
        store   s5, 07              ; 存储b *b 的高字节
        ; 计算a *a + b *b
        fetch   data, aa_lsb        ; 得到a *a 的低字节
        add   data, s6              ; 将a *a 和b *b 的低字节相加
        store   data, aabb_lsb      ; 存储a *a + b *b 的低字节
        fetch   data, aa_msb        ; 得到a *a 的高字节
        addcy   data, s5            ; 将a *a 和b *b 的高字节相加
        store   data, aabb_msb      ; 存储a *a + b *b 的高字节
        load   data, 00             ; 清除数据，但是保留进位
        addcy   data, 00            ; 从前面的操作中得到进位输出
        store   data, aabb_cout     ; 存储a *a + b *b 的进位输出
        return
; --------------------------------------------------------------------------------
; 程序：muit_soft
;   功能：使用移位叠加算法的8bit 无符号乘法
;   输入寄存器：
;     s3 : 被乘数
;     s4 : 乘数
;   输出寄存器：
;     s5 : 结果高字节
```

```
;      s6：结果低字节
;   临时寄存器：i
; ------------------------------------------------------------------------------------------
mult_soft：
        load  s5, 00          ; 清除s5
        load  i, 08           ; 初始化循环指针
mult_loop:
        sr0  s4               ; 将最低位移位至进位
        jump nc, shift_prod   ; 最低位是0
        add  s5, s3           ; 最低位是1
shift_prod:
        sra  s5               ; 将高字节右移,
                              ; 进位移至最高位，最低位移至进位
        sra  s6               ; 将低字节右移,
                              ; 将s5 的最低位移至s6 的最高位
        sub  i, 01            ; 循环指针递减
        jump nz, mult_loop    ; 重复直到i =0
        return
```

16.4.2　程序文件

汇编语言程序的编写是一个单调乏味的过程。标识符的使用和好的注释文本可使代码清晰并能减少很多不必要的错误，并且有助于后续校对和维护。对于 KCPSM3 汇编语言，我们可以使用 constant 命令定义一个标识符来代表常量、存储地址或端口 id，使用 namereg 命令定义寄存器变量标识符。

示例 16.2 列出了一个典型的主程序文件头，它包含以下几部分：

- 程序概述：概括描述程序要实现的功能、实现方式和 I/O(输入输出)端口；
- 常量声明：声明程序定义的符号常量；
- 数据 RAM 地址标识：声明程序定义的 RAM 地址标识符；
- 寄存器定义：声明程序定义的寄存器标识符；
- 端口定义：声明程序定义的端口；
- 程序调用层次说明：阐明程序调用关系和子程序功能。

这些标识符和命令对最终的机器语言代码没有任何影响，在汇编语言代码进行编译时它们被实际所指代的常量值替换。然而使用标识符能极大地提高汇编代码的可读性，减少不必要的错误。下面的这段代码进一步阐述了标识符和注释文

本的影响。这段代码的目的是为变量 a、b 和 c 赋值，并将它们存储到数据 RAM 相应的位置。存储地址由 UART 的输入，即字母 a、b 或 c 的 ASCII 码指定。下段代码使用了标识符和适当的注释：

```
; 常量别名
    constant  ASCII_a, 61   ; a 的ASCII 码
    constant  ASCII_b, 62   ; b 的ASCII 码
    constant  ASCII_c, 63   ; c 的ASCII 码
; 数据RAM 地址别名
    constant  a_addr, 02
    constant  b_addr, 04
    constant  c_addr, 06
; 寄存器别名
    namereg  s0, data      ; 临时数据寄存器
    namereg  sl, addr      ; 临时地址寄存器
    namereg  sF, sw_in     ; 转换器输入
; 端口别名
    constant  sw_port, 01        ; 转换器输入
    constant  uart_rx_port, 02   ; UART 输入
; 带有别名的汇编码
    ; 得到输入
    input  sw_in, sw_port        ; 得到转换器
    input  data, uart_rx_port    ; 得到字符
    ; 检查收到的字符
    compare  data, ASCII_a       ; 检查ASCII a
    jump nz, chk_ascii_b         ; 错误，检查下一个
    store  sw_in, a_addr         ; 正确，将a 存入数据RAM 中
    jump  done
chk_ascii_b:
    compare  data, ASCII_b       ; 检查ASCII b
    jump nz, chk_ascii_c         ; 错误，检查下一个
    store  sw_in, b_addr         ; 正确，将b 存入数据RAM 中
    jump  done
chk_ascii_c :
    compare  data, ASCII_c       ; 检查ASCII c
    jump nz, ascii_err           ; 错误
```

```
    store   sw_in, c_addr           ; 正确，将b 存入数据RAM 中
    jump    done
ascii_err :
    …
done :
    …
```

如果我们使用实际数值并且去掉注释，代码变为

```
; 没有别名和注释的汇编码
    input   sf, 01
    input   S0, 02
    compare   S0, 61
    jump nz, addrl
    store   sf, 02
    jump    addr4
addrl :
    compare   S0, 62
    jump nz, addr2
    store   sf, 04
    jump    addr4
addr2 :
    compare   S0, 63
    jump nz, addr3
    store   sf , 06
    jump    addr4
addr3 :
   …
addr4 :
   …
```

尽管这段代码与前述代码的功能是一样的，但是对于理解、调试和修改来说相当困难。

16.5　汇编代码处理

PicoBlaze-based 设计流程在 15. 4 节进行了回顾。汇编代码设计完成后，经过步骤 3，代码进行编译转换为机器指令。在步骤 4 中，指令级别的仿真也能验

证代码的正确性。在本章节对这两个步骤及直接下载过程（步骤 9）进行详细的论述讲解。

Xilinx 为步骤 3 中的编译过程提供了编译器 KCPSM3，并且为步骤 9 提供了下载应用程序。用户可以从 Xilinx 网站下载源程序、PicoBlaze 处理器的 HDL 代码和相关的文档模板。Mediatronix 公司提供的 PBlazeIDE 程序可进行步骤 4 中的指令级别仿真，也可被用作编译器。用户可从 Mediatronix 的网站下载 PBlazeIDE 软件。

16.5.1 KCSPM3 编译

编译软件将代码指令翻译为只有 0 和 1 的机器指令，并且用实际值替换标识符和分支地址代号，机器指令然后被下载到微处理器的指令存储器。因 PicoBlaze 嵌入到 FPGA 内部，指令存储 ROM 实际成为了一个存储编译后代码的 HDL ROM 模块。此 ROM 将在顶层 HDL 代码、PicoBlaze 综合和 I/O 端口电路中进行示例说明。

Xilinx 提供了 KCPSM3 编译器，它是一个在 DOS 环境下运行、使用命令行进行操作的软件。KCPSM3 基本上采用汇编程序，同时提供必要的模板文件，并生成针对所述指令 ROM 中的 HDL 代码。编译一个汇编代码的过程如下：

1）创建一个工程目录，并且将 kcpsm3.exe、ROM_form.vhd、ROM_form.v 和 ROM_form.coe 文件复制到目录下，后 3 个文件为 KCPSM3 的模板文件。

2）创建汇编程序并保存为扩展名为 .psm 的文本文件，任何基于 PC 的编辑器，如 Notepad，均能用来完成此任务；

3）调用 DOS 命令窗口：选择“开始”→“程序”→附件→命令提示符，在 DOS 命令窗口中，进入到工程目录下；

4）输入 kcpsm3 myfile.psm 命令运行程序；

5）如果有语法错误，改正并重新编译；

6）编译成功后，文件会包含指令 ROM，myfile.v 文件生成。

除 HDL 文件之外，KCPSM3 还产生了用于 block RAM 初始化等功能的文件。扩展名为 .hex 的文件用于 JTAG 下载，在 16.5.3 节将对此做进一步论述。扩展名为 .fmt 的文件为 .psm 文件的重定格式，用于打印。

16.5.2 PBlazeIDE 仿真

正如名字所示，指令级仿真指按指令一条条执行来模拟 PicoBlaze 系统的操作。PBlazeIDE 软件可用于实现此功能。PBlazeIDE 是一个 Windows 操作系统下的综合开发环境，包含文本编辑器、编译器和指令级仿真器。PBlazeIDE 使用的指令在存储方式和说明中有一些差异，详细情况已在 15.5 节进行了论述。这样，

用 KCPSM3 软件编写的代码并不能直接用于 PBlazeIDE，存储方式的差异在表 16-1 中进行了总结，命令的差异在表 16-2 中进行了总结。值得注意的是，PBlazeIDE 编译器中常量对于十进制和十六进制格式均支持，十六进制数据以 $ 符号开头，如 $ 1A。

表 16-1　KCPSM3 和 PBlazeIDE 存储方式的差异

KCPSM3	PBlazeIDE	KCPSM3	PBlazeIDE
addcy	addc	output sX，(sY)	out sX，sY
subcy	subc	output sX，KK	out sX，$ KK
compare	comp	return	ret
store sX，(sY)	store sX，sY	returni	reti
fetch sX，(sY)	fetch sX，sY	enable interrupt	eint
input sX，(sY)	in sX，sY	disable interrupt	dint
input sX，KK	in sX，$ KK		

表 16-2　KCPSM3 和 PBlazeIDE 指令举例

功能	KCPSM3	PBlazeIDE
代码位置	address 3FF	org $ 3FF
常量	constant MAX，3F	MAX equ $ 3F
寄存器别名	namereg addr，s2	addr equ s2
端口别名	constant in_port，00	in_port dsin $ 00
	constant out_port，10	out_port dsout $ 10
	constant bi_port，0F	bi_port dsio $ 0F

KCPSM3 代码用于 PBlazeIDE 软件仿真的过程如下：

1）用 KCPSM3 编译汇编语言代码；

2）运行 PBlazeIDE；

3）选择“Settings”→“PicoBlaze 3”，此操作指定 PicoBlaze 使用的为版本 3，Spartan-3 使用的为该版本；

4）选择“File”→“Import”，跳出一个对话框，选择相应的 .fmt 文件，“Import”选项的作用是将 KCPSM3 代码转换为 PBlazeIDE 代码，格式转换程序应用非常简单，但转换后的文件有时还需要一些小的手动辅修。

5）手动设定 dsin、dsout 和 dsio 命令设置 I/O 端口，当这些命令中的任一个被使用时，一个端口指示器将被添加到仿真屏幕显示这个端口的行为；

6）进入仿真模式，选择“Simulate”→“Simulate”执行仿真。

7）如果汇编代码需要修改，只能在 PBlazeIDE 外部完成，关闭当前文件，调用外部编辑器编辑原始的 . psm 文件，保存，然后从步骤 1 重新开始。如果文件在 PBlazeIDE 内进行了编辑，能逆转换到 KCPSM3 代码。

典型的仿真截图见图 16-2，仿真器在中间窗口显示汇编代码，高亮显示即将执行的下一条指令。指令地址、指令代码和断点被显示到代码旁侧。PicoBlaze 目前将其显示在左侧，包含 flag 状态、寄存器的内容和 data RAM 的内容。程序计数器的值和堆栈指针以及一些执行状态统计表显示在底栏。

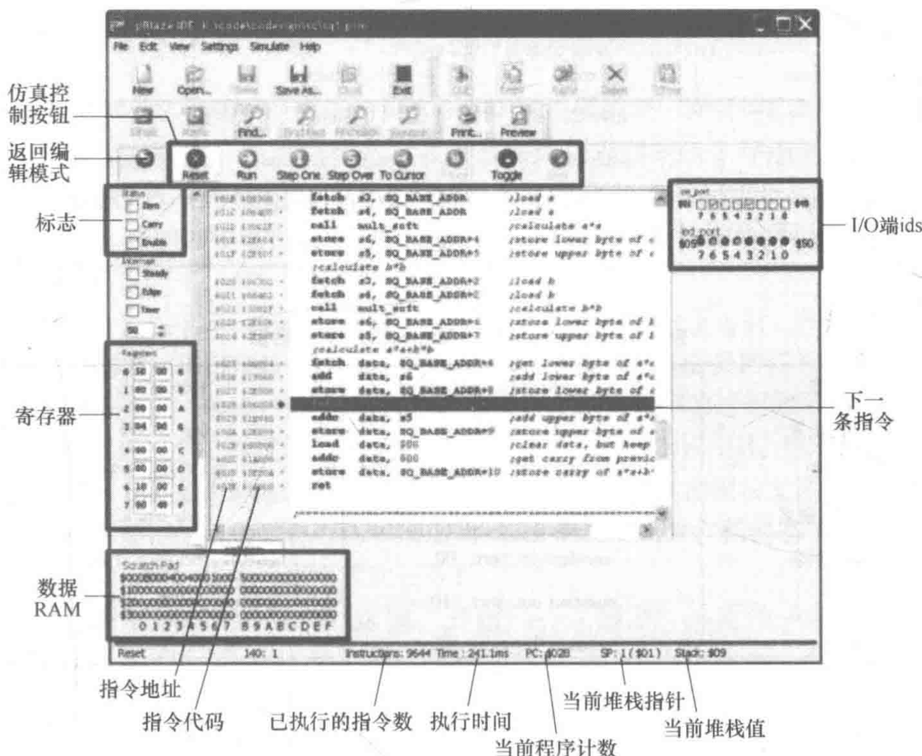


图 16-2　PBlazeIDE 仿真截图

由 dsin、dsout 和 dsio 命令产生的 I/O 端口显示在右侧。有一个输入端口、转换器、输出端口、LED 在目前的这个屏幕上。由于 PBlazeIDE 无 I/O 行为信息，在仿真中输入端口数据必须手动输入和更改。

仿真中，汇编程序执行方式有连续执行、按指令步步执行、执行到断点处暂停等执行方式。仿真方式可通过 Simulate 菜单命令或顶部图标选项进行控制：

- Reset：清除程序计数器和堆栈指针；
- Run：连续运行程序至断点处；
- Single step：执行一条指令；

- Step over：全部执行，有一个调用指令及执行整个子程序；
- Run to cursor：执行程序到目前的指针位置；
- Pause：暂停仿真；
- Toggle breakpoint：在目前的指针位置设置或清除断点；
- Remove all breakpoints：清除所有断点。

16.5.3 JTAG 重载

指令 ROM HDL 代码产生之后，我们可以继续15.4节所述的步骤6和8，综合全部代码并下载配置文件到 FPGA 芯片。值得注意的是，每次代码修改后综合码流必须重新下载。

由于综合是一个复杂的过程，它需要很长的计算时间。当I/O的设置固定之后，每次代码修改后不需要重新综合整个电路。我们可通过 FPGA 的 JTAG 接口将机器语言代码重载到 ROM，该 ROM 采用 block RAM 实现。这与图15-4虚线所示的步骤9一致。基本的流程如下：

1）用包含 JTAG 接口电路的模板替换原始的 ROM 模板；

2）使用 KCPSM3 编译汇编代码；

3）综合顶层 HDL 代码，并对 FPGA 芯片编程；

4）如果汇编子程序做了修改，需重新进行综合，生成扩展名为 .hex 的文件；

5）使用 Xilinx 工具将 .hex 文件内嵌到 JTAG 编程文件中，并由 JTAG 接口下载到 FPGA 的 block RAM 中。

详细过程及相关程序和模板可在下载的 KCPSM 文件中的 JTAG-loader 目录下找到。

16.5.4 PBlazeIDE 编译

正如前面论述的，PBlazeIDE 是一个包含编译器和编辑器的集成软件。PBlazeIDE 还能产生指令 ROM HDL 文件。但这个文件只能是 VHDL 文件。由于 Xilinx IST 支持混合语言仿真，这个文件仍可以被例化到顶层 Verilog 模块。详细过程在 IST 用户手册中有介绍。

为了得到指令 ROM 文件，我们将 vhdl 指令包含到汇编代码中。语法格式为

vhdl " ROM_form.vhd", " rom_target.vhd", " rom_entity_name"

这3个参数分别指定了 VHDL 模板文件（已在16.5.1节论述）、产生的 ROM VHDL 文件的名称和 VHDL 文件中实体的名称。值得注意的是，由于 PBlazeIDE 未产生 .hex 文件，16.5.3节讲述的重载过程在此不能直接使用。

16.6 PicoBlaze 综合

为指令 ROM 产生 HDL 文件后，可将其与 PicoBlaze 联合起来，将 FPGA 芯片上的整个系统进行综合。和普通微处理器不同，PicoBlaze 没有内置的 I/O 外围设备。I/O 端口是根据需要创建和定制的，相应电路在 HDL 代码中有描述。由于本章重点是汇编程序设计，我们使用一个简单的 I/O 配置端口来进行综合，它仅包含一个转换器输入端口和一个 LED 输出端口。更多复杂的 I/O 接口将在第 17 章和第 18 章进行论述。

设计的顶层模块结构如图 16-3 所示，它包含 PicoBlaze 处理器 KCPSM3、指令 ROM 和一个寄存器。寄存器作为 8 个 LED 数据的缓存。当 PicoBlaze 执行 output指令时，将数据放置到 out_port 端口并置 write_strobe 信号有效，将数据存储到寄存器。sw 信号连接到 in_port 端口，当 PicoBlaze 执行 input 指令时，输入端口获取 sw 信号的值并存储到内部寄存器。示例 16.3 列出了相应的 HDL 代码。它包含 PicoBlaze 处理器和指令 ROM 的例化及一段输出缓存相关的代码。KCPSM3 模块是 PicoBlaze 处理器，它的代码存储在同为该名的 HDL 文件中，sio_rom 模块是前面产生的指令 ROM 配置文件。

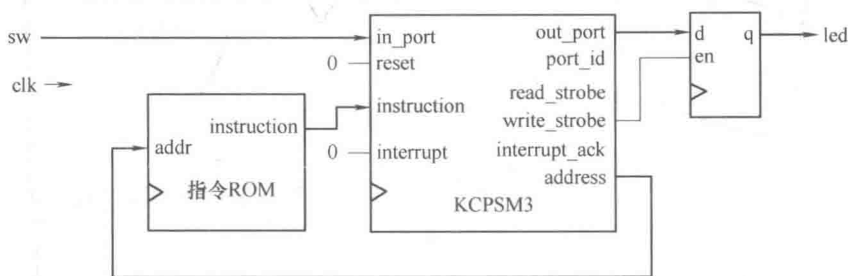


图 16-3 具有简单 I/O 接口的 PicoBlaze

示例 16.3 PicoBlaze 实现一个简单的 I/O 配置

```
module pico_sio
(
  input wire clk, reset,
  input wire [7: 0]     sw,
    output wire [7: 0]    led
);
//信号声明
```

```
//KCPSM3/ROM 信号
  wire [9: 0] address;
wire [17: 0] instruction;
wire [7: 0] port_id , in_port , out_port ;
wire write_strobe;
//寄存器信号
  reg [7: 0] led_reg;
// 实体
//..................................................
//KCPSM 和ROM 实例
  //..................................................
kcpsm3 proc_unit
      (.clk (clk), .reset (reset), .address (address),
     .instruction (instruction), .port_id (),
     .write_strobe (write_strobe), .out_port (out_port),
              .read_strobe (), .in_port (in_port), .interrupt (1'bO),
     .interrupt_ack ()) ;
sio_rom rom_unit
          ( .clk (c1k) , . address (address),
          .instruction (instruction));
//..................................................
//输出接口
//..................................................
always Q (posedge clk)
  if (write_strobe)
     led_reg < = out_port ;
assign led = led_reg;
//..................................................
//输入接口
//..................................................
     assign in_port = sw;
endmodule
```

16.7 文献备注

本章的文献备注内容和第 15 章相似。基于 JTAG 端口的编译后代码重载过程在 Kris Chaplin 和 Ken Chapman 的著作《PicoBlaze JTAG Loader Quick User Guide》中进行了论述。该文章在下载的 KCPSM 文件 JTAG-loader 目录下可找到。

16.8 实验

16.8.1 有符号数乘法运算

示例 16.1 中子程序将输入作为无符号整型数据处理。更改子程序进行有符号数据的乘法运算，输入和输出均作为有符号数处理，并通过仿真进行验证。

16.8.2 多字节乘法运算

示例 16.1 程序中输入数据位宽为 8 位，一些应用可能需要更高的精度。我们想将程序扩展，使输入为 16 位无符号数，一个操作数现在需要两个寄存器，输出结果需要 4 个寄存器。改进程序并且进行仿真验证。

16.8.3 循环位移功能

PicoBlaze 仅能位移或循环位移 1 位，“桶形”位移功能可实现多位的位移。此功能需有 3 个输入寄存器，第一个寄存器存放将被位移的数据，第二个寄存器在 0 ~ 7 范围内设定移的位数，第三个寄存器指示操作类型—左移、右移、循环左移或循环右移。改进程序并且进行仿真验证。

16.8.4 高低位互置功能

高低位互置功能是指将输入数据的高位和低位倒转互换。如果输入数据为“01010011”，则输出数据为“11001010”。我们可以使用 8bit 的转换器输出数据作为输入，而用 8bit 的离散 LED 作为输出。编写代码，进行仿真，并获得指令 ROM 文件，生成顶层 HDL 代码，将整个系统进行综合、验证。

16.8.5 二进制码至 BCD 码转换

二进制码至 BCD 码转换在 6. 3. 3 节进行了介绍，此功能可使用汇编代码实现。设定输入为 8bit 二进制数，输出为两个阿拉伯数字的 8bit BCD 码。如果输入大于 99，输出设置为溢出标志“11111111”。我们可以使用 8bit 的转换器输出

数据作为输入，而用 8bit 的离散 LED 作为输出。编写代码，进行仿真，并获得指令 ROM 文件，生成顶层 HDL 代码，将整个系统进行综合、验证。

16.8.6　BCD 码至二进制码转换

重复实验 16.8.5，编写代码，综合出电路，实现 BCD 码到二进制码的转换。

16.8.7　心跳电路

“心跳电路”在实验 4.7.4 进行了讲解，我们也可以使用 8bit 的离散 LED 实现一个类似的模型。编写代码，进行仿真，并获得指令 ROM 文件，生成顶层 HDL 代码，将整个系统进行综合、验证。

16.8.8　旋转闪亮 LED 电路

我们想设计一个电路，将 4 种 LED 图样“00000001”、“00000011”、“00001111”和“00001101”用 4 种不同的速度进行左移或右移。图样、方向和旋转速度可通过 8bit 的转换器（只使用 5bit）进行选择。速度应进行合理选择以使 4 种图样全能展现出来。编写代码，进行仿真，并获得指令 ROM 文件，生成顶层 HDL 代码，将整个系统进行综合、验证。

16.8.9　离散 LED 调光器

PWM 和 LED 调光器的概念在实验 4.7.2 已进行了介绍。在本实验中，我们想使用 8 个离散的 LED 显示不同级别的亮度。这可通过改变 LED 的“on”分数值来实现，8 个 LED 的“on”分数值将为 8/8、7/8、6/8、…、1/8。编写代码，进行仿真，并获得指令 ROM 文件，生成顶层 HDL 代码，将整个系统进行综合、验证。

第 17 章　PicoBlaze I/O 接口

17.1　简介

使用一个规范的由多种内置 I/O 外围接口（例如 UART、SPI、定时器等）组成的微控制器芯片来完成与外部环境的交互。当启动一个新的开发过程时，我们需要依据 I/O 在应用方面的需求去选择微控制器芯片，许多情况下还需要使用额外的芯片去实现一些不太常见的功能。

不同于常规微控制器，PicoBlaze 中并不包含内置的外设 I/O，PicoBlaze 中只有一个简单的 I/O 接口。PicoBlaze 的外设 I/O 需要根据需要来定制。PicoBlaze 使用 I/O 接口来在内部寄存器和 I/O 端口之间传输数据。I/O 接口包含以下信号：

- port_id：8bit，I/O 端口 id（或端口地址）；
- in_port：8bit，数据输入端口，PicoBlaze 使用 input 指令从该端口获取数据；
- out_port：8bit，数据输出端口，PicoBlaze 使用 output 指令来通过该端口输出数据；
- read_strobe：1bit，标志信号，该信号在 input 指令的第二个时钟周期置为有效；
- write_strobe：1bit，标志信号，该信号在 output 指令的第二个时钟周期置为有效。

尽管仅有两个 8bit 端口用于输入、输出数据，但 port_ id 可以被用于区别不同的外设，因此 PicoBlaze 可以最多支持 256（即 2^8）个 I/O 端口。

在本章接下来的部分中，我们将仔细研究 PicoBlazeI/O 的时序，通过为第 16 章中所述电路增加一系列外设接口，我们将进一步说明如何进行 I/O 接口的开发。

17.2　输出端口

17.2.1　output 指令及时序

通过 output 指令向输出端口写数据，有两种格式：

output sX, (sY)

output sX, 端口名

第一种格式中，端口 id 存储于 sY 寄存器中。第二种格式中，端口 id 通过端口名来直接指定，端口名为一个两位的十六进制数或者一个预先定义好的符号常量。输出数据存储于 sX 寄存器中。

输出指令的时序图，

output s0, 02

在图 17-1 中最上部的五行显示。前文说过每个 PicoBlaze 指令需要两个时钟周期，在指令被执行的两个时钟周期内，s0 的内容会被传输至输出端口上，同时端口 id 置为 02。write_strobe 信号在第二个时钟周期被置为有效，该信号可以作为在输出寄存器中保存数据的使能标志或用于对外设进行初始化。

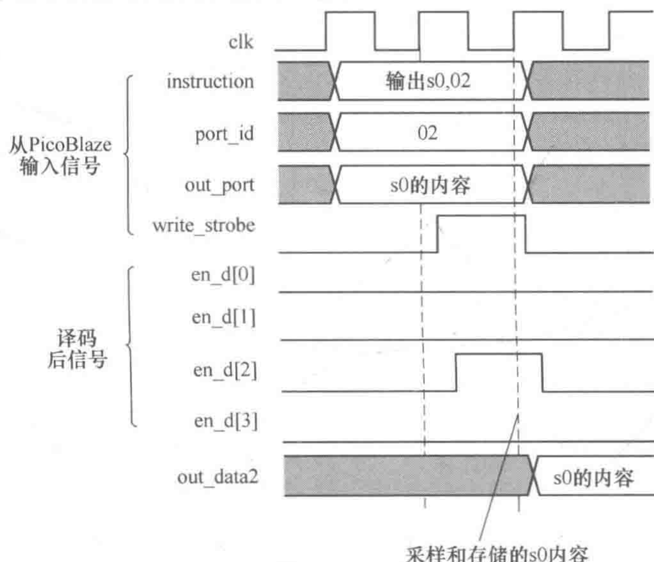


图 17-1　输出结构时序图

17.2.2　输出接口

PicoBlaze 和输出外设之间的输出接口通常包含有解码电路和输出缓存，输出缓存通常由寄存器阵列构成。解码电路解析端口 id，并据此生成一个使能信号。输出指令执行后，数据会存储到指定的缓存中。

我们以四输出的 PicoBlaze 接口为例来说明这种结构。指定 00_{16}、01_{16}、02_{16} 和 03_{16}作为其端口 id。可见，端口地址的 6 个 MSB 是一样的，仅用两个 LSB 去区分一个端口。方框图如图 17-2 所示。其中核心为解码电路，表 17-1 为解码电路功能真值表，它是一个 2-2 解码器。在输出指令的第二个时钟周期，write-

strobe 被置为有效，en_d 信号中的其中 1 位也置为有效。en_d 为单时钟周期使能信号，每一位对应相应的寄存器。当某一位置为有效时，相应的寄存器便从输出端口中获取数据。

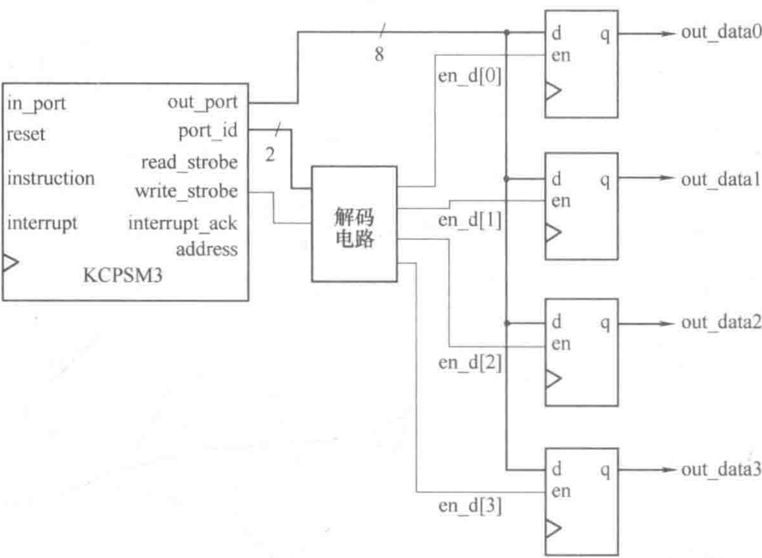


图 17-2　四输出寄存器的输出解码图

表 17-1　解码电路真值表

write_strobe	输入 端口 id[1]	端口 id[0]	输出 en_d
0	—	—	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

该指令的解码时序图

output s0，02

在图 17-1 底部显示。在输出指令的第二个时钟周期，en_d[2]置为有效，输出接口中的数据在下一个时钟上升沿存储在 en_d[2]所对应的缓存中。

一旦了解了基本的操作，我们就能据此来编写 HDL 代码。代码段为

```
always @ *
  if (write_strobe)
    case (port_id[1:0])
    2'b00: en_d =4'b0001;
    2'b01: en_d =4'b0010;
    2'b10: en_d =4'b0100;
    2'b11: en_d =4'b1000;
    endcase
  else
    en_d =4'b0000;
```

这个设计非常的概括，可以扩展至任意数量输出端口。

端口地址的选择较为随意。我们在上个例子中使用二进制编码的方式来定义端口地址。如果输出端口的数量小于 8，独热码可以被用于简化解码电路。例如，我们可以定义上面的 4 个端口 id 为：01_{16}（如，00000001_2）、02_{16}（如，00000010_2）、04_{16}（如，00000100_2）和 08_{16}（如，00001000_2）。解码逻辑可以简化为

```
always @ *
  if (write_strobe)
    en_d = port_id[3:0];
  else
    en_d =4'b0000;
```

值得注意的是，如果仅有一个输出端口，那么就不需要解码逻辑。Write-strobe 信号可以连接到寄存器使能信号端，如图 16-3 所示。

在 16.4.2 节中讨论的，一个好的方法是使用符号化命名的方式去定义 I/O 端口，并且在头文件中定义二进制地址。例如，输出端口地址分配可以声明如下：

```
-----------------------------output port definitions-----------------------------
constant out-port-a, 00
constant out-port-b, 01
constant out-port-c, 02
constant out-port-d, 04
```

如果地址分配变更，我们仅更新头文件即可，这样就保证了其余代码的完整性。使用清晰的头文件，同样可以令我们轻松地更改同伴开发的 HDL 文件的端口 id。

17.3 输入端口

17.3.1 输入指令和时序

输入指令从输入端口读取数据。与输出指令相似，其具有两种格式：

input sX,(sY)

output sX,port_name

sY 寄存器或端口名用于指示读端口 id。读取的数据被存储在 sX 寄存器中。

输入指令的时序图，

input s0，02

在图 17-3 中显示。当指令被执行，端口 id 置为 02。两个时钟周期后，输入端口中的数据会在时钟的上升沿被采样，并存储在 s0 寄存器中。外部电路必须确保输入数据在采样时钟沿保持稳定，从而避免一些时序违规。

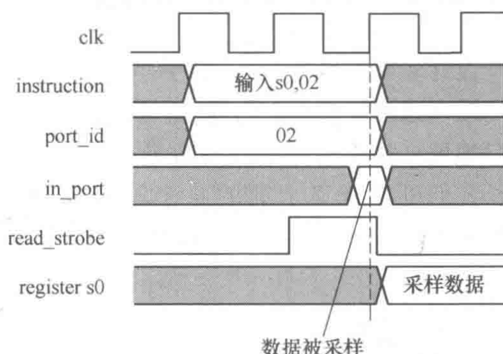


图 17-3 输入指令时序图

如同输出指令中一样，read-strobe 信号在第二个时钟周期的上升沿置为有效。Read-strobe 信号的功能不是很明显，这部分会在此后的章节中讨论。

17.3.2 输入接口

PicoBlaze 和输入外设之间的输入接口通常包含多路选择器，多路选择器使用 port_id 作为选择信号来为 in_port 选择所需的数据。有时，输入接口也需要类似输出接口中的解码电路来实现数据访问。

在输入接口设计的过程中，输入端口可以被分为连续访问或单独访问端口。对于一个连续访问端口来说，数据是连续出现的，类似于 16.4.1 节中的选择输入。而单独访问端口的数据有效性是由单独的无关事件所触发的，例如通过 UART buffer 接收一个字符。8.2.4 节中讨论的标志寄存器和 buffer 就属于这种类别。数据被获取后，我们需要将其从 buffer 中移出，以防止相同的数据被重复读取。通常用单周期的信号去清空标志寄存器或将一个字从 FIFO buffer 中移出。

连续访问端口的接口仅包括一个多路选择器电路。以一个包括 4 个此种端口的接口为例，方框图如图 17-4 所示。

单独访问端口的接口需要一个状态机在输入指令结束时将获取的数据从

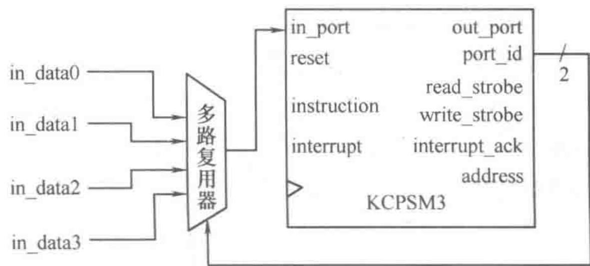


图 17-4 四连续访问端口方框图

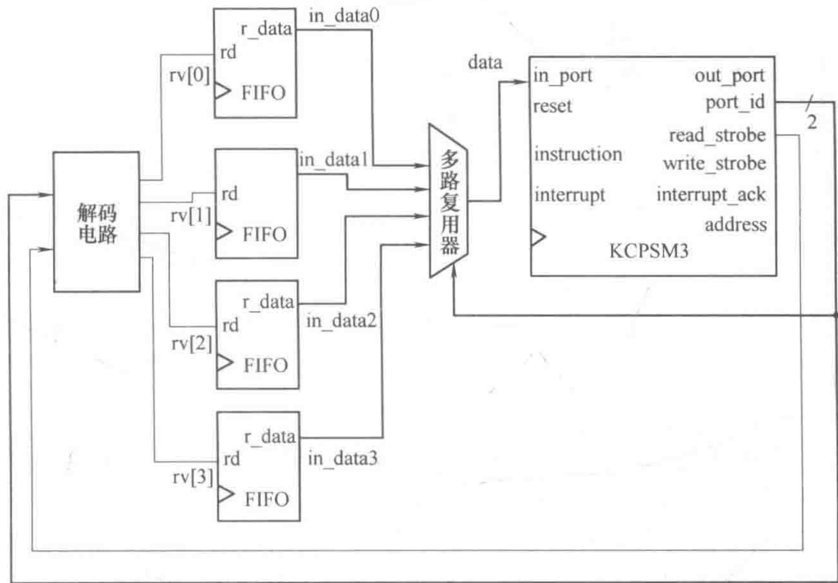


图 17-5 四单独访问端口方框图

buffer 中移出。可以通过解码电路来对 port_id 和 read-strobe 进行解码。解码电路同输出接口中的解码电路是相同的，只是将 write-strobe 替换成 read-strobe。解码器的输出可以看做单时钟周期的“清除”信号。以一个包含 4 个 FIFO 的接口为例。完整的解码和多路电路框图如图 17-5 中显示。信号 rv 为解码得到的“清除”信号。在输入指令结束后，该 4bit 信号中的 1bit 将置为有效，相应的 FIFO 会执行一次读操作，第一个字将从 buffer 中移出。假设端口 id 为 00_{16}、01_{16}、02_{16} 和 03_{16}。则接口的 HDL 代码为

```
//复用电路
always @ *
  case (port_id[1:0])
    2'b00 : data = in_data0;
```

```
      2'b01: data = in_data1;
      2'b10 : data = in_data2;
      2'b11 : data = in_data3;
    endcase
//解码电路
always @ *
    if (read_strobe)
      case (port_id[1:0])
        2'b00 : rv =4'b0001;
        2'b01 : rv =4'b0010;
        2'b10 : rv =4'b0100;
        2'b11 : rv =4'b1000;
      endcase
    else
      rv =4'b0000;
```

在实际应用中，输入接口通常既包含连续访问接口也包含单独访问接口。解码电路只在单独访问接口中才有。

17.4　包含开关输入和 7 段 LED 显示接口的二次方计算程序

为了进一步说明 PicoBlaze I/O 接口的架构，我们以第 16 章中的二次方计算程序为例，添加更多的通用外设。二次方计算程序的功能是计算 a^2+b^2 的值，其中 a 和 b 为 8 位无符号整型数。

我们使用 8bit 开关和一个按钮来输入数值 a 和 b。当按下按钮会生成一个单时钟周期的标志信号。此标志信号指示当前开关的数值需要被导入。数值 a 和 b 会被交替导入；例如：第一次按下按钮会导入 a，第二次按下就会导入 b，第三次又会导入 a，以此类推。使用另外一个按钮来清空 PicoBlaze 的数据 RAM 和相关寄存器。

我们使用 4 个七段 LED 来显示输入和计算结果。LED 被设定为 4 个十六进制数。因为 a^2+b^2 数值范围最大为 17bit，最左端的 LED 的小数点用于指示 MSB。开关的最低 3 个 bit 选择 a、b、a^2、b^2 或 a^2+b^2 哪一个被显示。

综上所述，接口包括以下内容：

- 开关：提供 a 和 b 的数值，选择 LED 显示的内容；
- 按钮 0：当按下时交替导入数值 a 和 b；
- 按钮 1：当按下时清空数据 RAM 及其相关的寄存器；

● 七段 LED：按 4 个十六进制数的方式显示所选择的 17bit 数值。

17.4.1 输出接口

原型开发板上的 4 个七段 LED 共享相同的输入引脚，因此我们需要设计一个时分复用电路。在基于 PicoBlaze 的设计中，复用电路既可以使用外部电路实现也可以通过软件程序实现。本节中我们选择外部电路的方式，这样能够简化汇编代码的开发，第 18 章中将会讨论软件程序实现的方法。在这里，我们可以使用 4.5.1 节中所讨论的 LED 时分复用电路。该电路使得时分复用的过程对于外部系统而言是透明的，外部系统所能看到的是 4 个相互独立的 7 段 LED。PicoBlaze 的输出接口方框图在图 17-6 中显示。接口包括 4 个 8bit 输出端口，每个端口对应一个 LED。

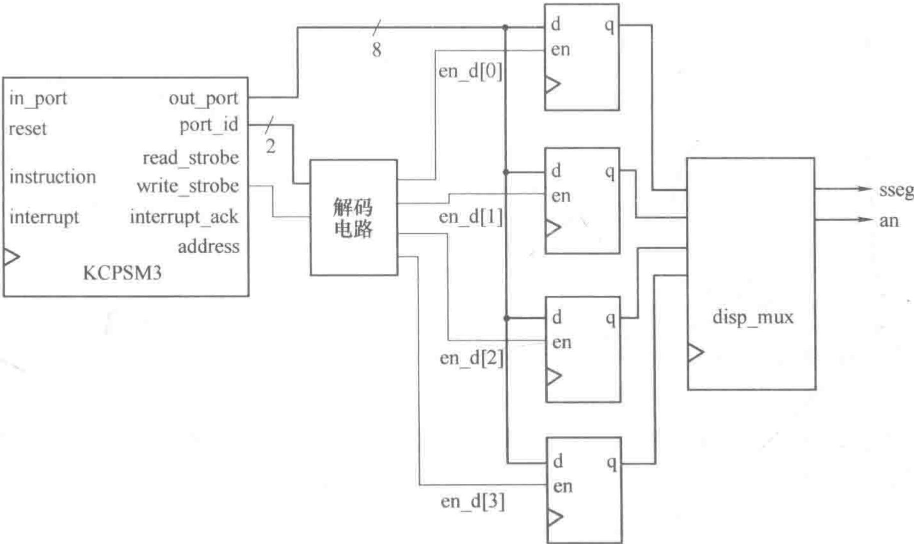


图 17-6 二次方电路的输出接口

在汇编代码中，4 个 LED 分别对应 PicoBlaze 数据 RAM 的 4 个地址，分别为 led0、led1、led2 和 led3。相应的代码段为

```
...
;data RAM address alias
constant led0,10
constant led1,11
constant led2,12
constant led3,13
...
```

```
;output port definitions
constant sseg0_port,00;7-seg led 0
constant sseg1_port,01;7-seg led 1
constant sseg2_port,02;7-seg led 2
constant sseg3_port,03;7-seg led 3
...
disp_led:
    fetch data,led0;
    output data,sseg0_port
    fetch data,led1;
    output data,sseg1_port
    fetch data,led2;
    output data,sseg2_port
    fetch data,led3;
    output data,sseg3_port
    return
```

17.4.2 输入接口

输入接口包含一个 8bit 开关和两个 1bit 按钮。前者为一个连续访问端口，其数值是持续有效的。后者为一个单独访问端口，因为按下一个按钮只会触发一个事件（向寄存器中导入一次数据，而非连续导入）。由于机械按钮会产生毛刺，因此需要使用去抖动电路来产生单时钟周期的脉冲激励。因为 PicoBlaze 的端口可以接收 8bit 数据，从两个按钮的输入可以整合为一组，作为一个独立的输入端口。输入接口的方框图在图 17-7 中显示。接口包括两个去抖动电路，一个 2 选 1 的多路选择器，一个解码电路，两个标志寄存器。两个标志寄存器的功能在 8.2.4 节中已讨论过。它们用于置位和复位“按钮触发事件”。当一个按钮被按下，去抖动电路的输出会将标志寄存器置位。在 PicoBlaze 发出输入指令前，该标志寄存器会一直保持原值。输入指令设置多路选择器的选择信号，从而将期望的数据传输至 PicoBlaze 的输入端口，并激活复位信号。为了便于讨论，我们将按钮 1 命名为 s 按钮（用来进行数值设置），按钮 0 命名为 c 按钮（用来复位数据 RAM）。

执行输入的伪代码为

```
; input the button flags
; if c = 1 then
;    call the clearing-ram routine
```

```
; if s = 1 then
;    input switch value
;    store it to data ram
;    toggle a/b address offset
```

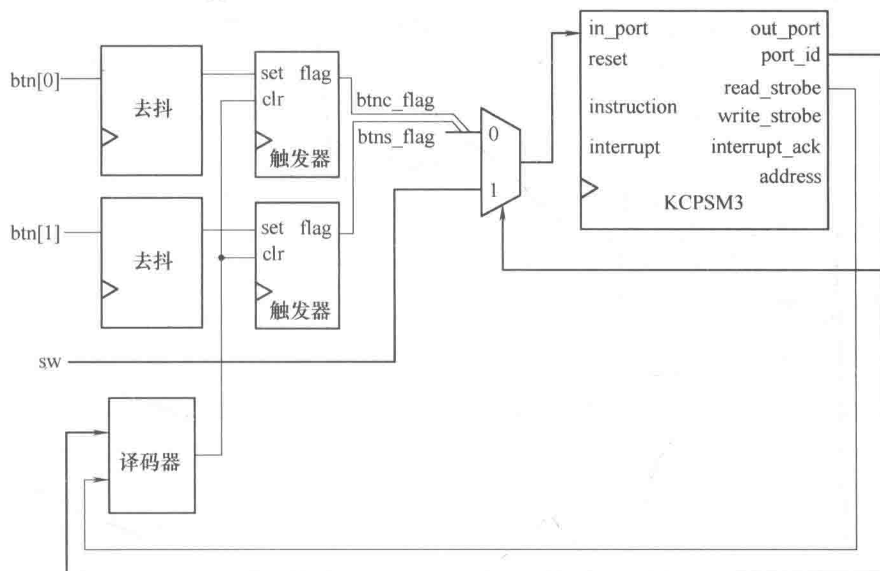


图 17-7 电路输入接口图

因为 s 按钮交替输入 *a* 和 *b*，我们使用一个全局寄存器 switch_ a_ b 来记录当前正在读取的数值是 *a* 还是 *b*。该寄存器的值同时也是数据 RAM 地址的偏移量，其值可以为 0 或 2，并且在 s 按钮被按下时锁存。相应的汇编代码子程序为

```
;input port definitions
constant rd_flag_port,00;2 flags (xxxxxxsc);
...
proc_btn:
   input s3,rd_flag_port;get flag
   ;check and process c button
   test s3,01;check c button flag
   jump z,chk_btns;flag not set
   call init;flag set,clear
   jump proc_btn_done
chk_btns:
   ;check and process s button
```

```
    test s3,02;check s button flag
    jump z,proc_btn_done;flag not set
    input data,sw_port;get switch
    load addr,a_lsb;get addr of a
    add addr,switch_a_b;add offset
    store data,(addr);write data to ram
    ;update current disp position
    xor switch_a_b,02;toggle between 00,02
  proc_btn_done:
    return
```

17.4.3 集成代码开发

在完成 I/O 接口设计后，我们可以开始设计汇编程序。本开发遵循第 16 章提出的分解法来进行，将主程序划分成若干子程序。主程序为

```
    call init;initialization
  forever:
    ;main loop body
    call proc_btn;check & process buttons
    call square;calculate square
    call load_led_pttn;store led patterns to ram
    call disp_led;output led pattern
    jump forever
```

完整代码在示例 17.1 中列出。

二次方子程序来自第 16 章，proc_btn 和 disp_led 子程序在之前的两个章节中已经进行过讨论。init 子程序用于实现系统的初始化，使用 loop 循环将 0’s 导入数据 RAM（即复位 RAM）并将 switch_a_b 寄存器置为 0（即读取 *a*）。load_led_pttn 子程序读取开关输入，从 RAM 中获取所需的数据，将数值转换成七段 LED 所需的格式，并将它们存入 RAM 相应的地址空间中。这些数据随后通过 disp-led 子程序写入输出端口。load-led-pttn 子程序包括 get-uppernibble 子程序和 get-lowernibble 子程序，用于生成 8bit 所对应的两个十六进制数字，hex-to-led 子程序将一个十六进制数字转换成相应的 7 段 LED 格式。

这个程序需要更多的存储空间。除了数据 RAM 和用于二次方计算程序的寄存器外，本程序还使用了一个新的全局寄存器 switch-a-b 去标志究竟是 *a* 还是 *b* 正在被读取。同时，数据 RAM 中有四字节的地址被标注为 led0、led1、led2 和 led3，这 4 个字节的存储空间用来存储 4 个 7 段 LED 的显示输出。

示例 17.1　包含开关和 7 段 LED 接口的二次方计算程序

```
;=====================================================
;square circuit with 7-seg LED interface
;=====================================================
;program operation:
;-read a and b from switch
;-calculate a * a + b * b
;-display data on 7-seg led
;=====================================================
;data RAM address alias
;=====================================================
constant a_lsb,00
constant b_lsb,02
constant aa_lsb,04
constant aa_msb,05
constant bb_lsb,06
constant bb_msb,07
constant aabb_lsb,08
constant aabb_msb,09
constant aabb_cout,0A
constant led0,10
constant led1,11
constant led2,12
constant led3,13
;=====================================================
;register alias
;=====================================================
;commonly used local variables
namereg s0,data;reg for temporary data
namereg s1,addr;reg for temporary mem & i/o port addr
namereg s2,i;general-purpose loop index
;global variables
;=====================================================
;port alias
```

```
;==================================================================
;--------------------------------------input                               port
definitions---------------------------------------------
    constant rd_flag_port,00;2 flags(xxxxxxsc):
    constant sw_port,01;8-bit switch
;--------------------------------------output                              port
definitions---------------------------------------------
    constant sseg0_port,00;7-seg led 0
    constant sseg1_port,01;7-seg led 1
    constant sseg2_port,02;7-seg led 2
    constant sseg3_port,03;7-seg led 3
;==================================================================
;main program
;==================================================================
;calling hierarchy
;
;main
;   -init
;   -proc_btn
;      -init
;   -square
;      -mult_soft
;   -load_led_pttn
;      -get_lower_nibble
;      -get_upper_nibble
;      -hex_to_led
;   -disp_led
;==================================================================
   call init;initialization
forever:
   ;main loop body
   call proc_btn;check & process buttons
   call square;calculate square
   call load_led_pttn;store led patterns to ram
   call disp_led;output led pattern
```

```
  jump forever
;=========================================================
;routine:init
;   function:perform initialization,clear register/ram
;   output register:
;      switch_a_b:cleared to 0
;   temp register:data,i
;=========================================================
init:
  ;clear memory
  load I,40;unitize loop index to 64
  load data,00
clr_mem_loop:
  store data,(i)
  sub i,01;dec loop index
  jump nz,clr_mem_loop;repeat until i=0
  ;clear register
  load switch_a_b,00
  return
;=========================================================
;routine:proc_btn
;   function:check two buttons and process the display
;   input reg:
;      switch_a_b:ram offset(0 for a and 2 for b)
;   output register:
;      s3:store input port flag
;      switch_a_b:may be toggled
;   temp register used:data,addr
;=========================================================
proc_btn:
  input s3,rd_flag_port;get flag
  ;check and process c button
  test s3,01;check c button flag
  jump z,chk_btns;flag not set
  call init;flag set,clear
```

```
    jump proc_btn_done
chk_btns:
    ;check and process s button
    test s3,02;check s button flag
    jump z,proc_btn_done;flag not set
    input data,sw_port;get switch
    load addr,a_lsb;get addr of a
    add addr,switch_a_b;add offset
    store data,(addr);write data to ram
    ;update current disp position
    xor switch_a_b,02;toggle between 00,02
proc_btn_done:
    return
;=====================================================
;routine:load_led_pttn
;   function:read 3 LSBs of switch input and convert the desired values to four led
patterns
;            and load then to ram
;            switch:000:a; 001:b; 010:a^2; 011:b^2; others: a^2 + b^2
;   temp register used:data,addr
;      s6:data from sw input port
;=====================================================
load_led_pttn:
    input s6,sw_port;get switch
    sl0 s6; *2 to obtain addr offset
    compare s6,08;sw > 100?
    jump c,sw_ok;no
    load s6,08;yes,sw error,make default
sw_ok:
    ;process byte 0,lower nibble
    load addr a_lsb
    add addr,s6;get lower addr
    fetch data,(s6);get lower byte
    call get_lower_nibble;get lower nibble
    call hex_to_led;convert to led pattern
```

```
  store data,led0
  ;process byte 0,upper nibble
  fetch data,(addr)
  call get_lower_nibble
  call hex_to_led
  store data,led2
  ;process byte 1,upper nibble
  fetch data,(addr)
  call get_upper_nibble
  call hex_to_led
  ;check for sw =100 to process carry as led dp
  compare s6,08;display final result?
  jump nz,led_done;no
  add addr,01;get carry addr
  fetch s6,(addr);s6 to store carry
  test s6,01;carry =1?
  jump z,led_done;no
  and data,7F;yes,assert msb (dp) to 0
led_done:
  store data,led3
  return
;==========================================================
;routine:disp_led
;   function:output four led patterns
;   temp register used: data
;==========================================================
disp_led:
  fetch data,led0
  output data,sseg0_port
  fetch data,led1
  output data,sseg1_port
  fetch data,led2
  output data,sseg2_port
  fetch data,led3
  output data,sseg3_port
```

```
  return
;==============================================================
;routine: hex_to_led
;  function: convert a hex digit to 7-seg led pattern
;  input register: data
;  output register: data
;==============================================================
hex_to_led:
  compare data,00
  jump nz,comp_hex_1
  load data,81;7-seg pattern 0
  jump hex_done
comp_hex_1:
  compare data,01
  jump nz,comp_hex_2
  load data,CF;7-seg pattern 1
  jump hex_done
comp_hex_2:
  compare data,02
  jump nz,comp_hex_3
  load data,92;7-seg pattern 2
  jump hex_done
comp_hex_3:
  compare data,03
  jump nz,comp_hex_4
  load data,86;7-seg pattern 3
  jump hex_done
comp_hex_4:
  compare data,04
  jump nz,comp_hex_5
  load data,CC;7-seg pattern 4
  jump hex_done
comp_hex_5:
  compare data,05
  jump nz,comp_hex_6
```

```
   load data,A4;7-seg pattern 5
   jump hex_done
comp_hex_6:
   compare data,06
   jump nz,comp_hex_7
   load data,A0;7-seg pattern 6
   jump hex_done
comp_hex_7:
   compare data,07
   jump nz,comp_hex_8
   load data,8F;7-seg pattern 7
   jump hex_done
comp_hex_8:
   compare data,08
   jump nz,comp_hex_9
   load data,80;7-seg pattern 8
   jump hex_done
comp_hex_9:
   compare data,09
   jump nz,comp_hex_a
   load data,84;7-seg pattern 9
   jump hex_done
comp_hex_a:
   compare data,0A
   jump nz,comp_hex_b
   load data,88;7-seg pattern a
   jump hex_done
comp_hex_b:
   compare data,0B
   jump nz,comp_hex_c
   load data,E0;7-seg pattern b
   jump hex_done
comp_hex_c:
   compare data,0C
   jump nz,comp_hex_d
```

```
    load data,B1;7-seg pattern C
    jump hex_done
comp_hex_d:
    compare data,0D
    jump nz,comp_hex_e
    load data,C2;7-seg pattern d
    jump hex_done
comp_hex_e:
    compare data,0E
    jump nz,comp_hex_e
    load data,B0;7-seg pattern E
    jump hex_done
comp_hex_f:
    load data,B8;7-seg pattern F
hex_done:
    return
;==========================================================
;routine: get_lower_nibble
;   function: get lower 4 bits of data
;   input register: data
;   output register: data
;==========================================================
get_lower_nibble:
    and data,0F;clear upper nibble
    return
;==========================================================
;routine: get_upper_nibble
;   function: get upper 4 bits of in_data
;   input register: data
;   output register: data
;==========================================================
get_upper_nibble:
    sr0 data;right shift 4 times
    sr0 data
    sr0 data
```

```
  sr0 data
  return
;==================================================
;routine: square
;  function: calculate a * a + b * b
;     data/result stored in ram started w/SQ_BASE_ADDR
;  temp register: s3, s4, s5, s6, data
;==================================================
square:
  ;calculate a * a
  fetch s3,a_lsb;load a
  fetch s4,a_lsb;load a
  call mult_soft;calculate a * a
  store s6,aa_lsb:store lower byte of a * a
  store s5,aa_msb;store upper byte of a * a
  ;calculate b * b
  fetch s3,b_lsb;load b
  fetch s4,b_lsb;load b
  call mult_soft;calculate b * b
  store s6,bb_lsb: store lower byte of b * b
  store s5,bb_msb;store upper byte of b * b
  ;calculate a * a + b * b
  fetch data,aa_lsb;store lower byte of a * a
  add data,s6;add lower byte of a * a + b * b
  store data,aabb_lsb;store lower byte of a * a + b * b
  fetch data,aa_msb;get upper byte of a * a
  addcy data,s5;add upper byte of a * a + b * b
  store data,aabb_msb;store upper byte of a * a + b * b
  load data,00;clear data,but keep carry
  addcy data,00;get carry from previous
  store data,aabb_cout;store carry of a * a + b * b
  return
;==================================================
;routine:mult_soft
;  function: 8-bit unsigned multiplier using
```

```
;           shift-and-add algorithm
;  input register:
;     s3: multiplicand
;     s4:multiplier
;  output register:
;     s5:upper byte of product
;     s6:lower byte of product
;  temp register: i
;================================================
mult_soft:
   load s5,00;clear s5
   load i,08;initialize loop index
mult_loop:
   sr0 s4;shift lsb to carry
   jump nc,shift_prod;lsb is 0
   add s5,s3;lsb is 1
shift_prod:
   sra s5;shift upper byte right,
         ;carry to MSB,LSB to carry
   sra s6:shift lower byte right,
         ;lsb of s5 to MSB of s6
   sub i,01;dec loop index
   jump nz,mult_loop;repeat until i=0
   return
```

17.4.4 HDL 代码开发

完整的 HDL 代码包含有 PicoBlaze 处理器，指令 ROM，图 17-7 所示的输入接口和外设以及图 17-6 所示的输出接口和外设。它们在示例 17.2 中列出。

示例 17.2　包含开关和 7 段 LED 接口的 PicoBlaze

```
module pico_btn
(
   input wire clk,reset,
   input wire[7:0] sw,
```

```
  input wire[1:0]btn,
  output wire[3:0] an,
  output wire[7:0]sseg
)
//信号声明
//KCPSM3/ROM 信号
wire[9:0] address;
wire[17:0] instruction;
wire[7:0] port_id,out_port;
reg[7:0] in_port;
wire write_strobe,read_strobe;
//I/O 端口信号
//输出使能
reg[3:0] en_d;
//4 位7 段LED 显示
reg[7:0] ds3_reg,ds2_reg,ds1_reg,ds0_reg;
//两个按钮
reg btnc_flag_reg,btns_flag_reg;
wire btnc_flag_next,btnc_flag_next;
wire set_btnc_flag,set_btns_flag,clr_btn_flag;
//实体
//=================================================
//I/O 模块
//=================================================
disp_mux disp_unit
(.clk(clk),.reset(reset),
.in3(ds3_reg),.in2(ds2_reg),.in1(ds1_reg),
.in0(ds0_reg),.an(an),.sseg(sseg));
debounce btnc_unit
(.clk(clk),.reset(reset),.sw(btn[0]),
.db_level(),.db_tick(set_btnc_flag));
debounce btns_unit
(.clk(clk),.reset(reset),.sw(btn[1]),
.db_level(),.db_tick(set_btns_flag));
//=================================================
```

```
//KCPSM 和ROM 实例
// ==================================================
kcpsm3 proc_unit
(. clk(clk),. reset(1'b0),. address(address),
. instruction(instruction),. port_id(port_id),
. write_strobe(write_strobe),. out_port(out_port),
. read_strobe(read_strobe),. in_port(in_port),
. interrupt(1'b0),. interrupt_ack());
btn_rom rom_unit
(. clk(clk),. address(address),
. instruction(instruction));
// ==================================================
//输出接口
// ==================================================
//输出端口id
//  0x00 :ds0
//  0x01 :ds1
//  0x02 :ds2
//  0x03 :ds3
// ==================================================
//寄存器
always @ (posedge clk)
  begin
    if(en_d[0])
      ds0_reg < =out_port;
    if(en_d[1])
      ds1_reg < =out_port;
    if(en_d[2])
      ds2_reg < =out_port;
    of(en_d[3])
      ds3_reg < =out_port;
  end
//解码电路使能信号
always @ *
  if(write_strobe)
```

```
    case(port_id[1:0])
      2'b00:en_d =4'b0001;
      2'b01:en_d =4'b0010;
      2'b10:en_d =4'b0100;
      2'b11:en_d =4'b1000;
    endcase
  else
    en_d =4'b0000;
// =====================================================
// 输入接口
// =====================================================
// 输入端口 id
//  0x00 :flag
//  0x01 :switch
// =====================================================
// 输入寄存器
always @(posedge clk)
  begin
    btnc_flag_reg  <= btnc_flag_next;
    btns_flag_reg  <= btns_flag_next;
  end
assign btnc_flag_next = (set_btnc_flag) ? 1'b1 :
                        (clr_btn_flag) ? 1'b0 :
                        btnc_flag_reg;
assign btns_flag_next = (set_btns_flag) ? 1'b1 :
                        (clr_btn_flag) ? 1'b0 :
                        btns_flag_reg;
// 信号清晰的解码电路
assign clr_btn_flag = read_strobe && (port_id[0] ==1'b0);
// 输入多路复用
always @ *
  case(port_id[0])
    1'b0: in_port = {6'b0,btns_flag_reg,btnc_flag_reg};
    1'b1: in_port = sw;
  endcase
```

```
endmodule
```

17.5 结合组合乘法器和 UART 控制器的乘法程序

本节中，我们会向前面的设计中加入两个 I/O 外设：一个是组合乘法器，可以用来加速乘法运算；另一个是 UART，提供一个与 PC 通信的链路。

17.5.1 乘法器接口

因为 PicoBlaze 不包括硬件乘法器，乘法由软件程序 mult-soft 实现。运用移位结合加法的算法去迭代实现 8bit 乘法器。最坏的情况下，该程序需要大约 60 条指令去实现。另一种方法是利用 Spartan-3 器件内置的组合乘法器去实现。

由于 PicoBlaze 没有提供任何机制去使用一个协处理器，乘法器必须被配置成一个 I/O 外设。我们可以创建一个拥有两个 8bit 操作数并会返回一个 16bit 乘积的 8bit 组合乘法器。为了实现此架构，PicoBlaze 接口需要两个额外的输出端口和 buffer 以输出两个操作数以及两个额外的的输入端口来接收 16bit 的乘积。汇编程序只需要将操作数输出至输出端口，然后从输入端口获取结果即可。代码为

```
;input port definitions
constant mult_prod0_port,03;multiplication product 8 LSBs
constant mult_prod1_port,04;multiplication product 8 MSBs
;output port definitions
constant mult_src_port,05;multiplier operand 0;
constant mult_src_port,06;multiplier operand 1;
...
mult_hard:
   output s3,mult_src0_port
   output s4,mult_src1_port
   input s5,mult_prod1_port
   input s6,mult_prod0_port
   return
```

需要注意的是，组合乘法器可以通过一条指令（即两个时钟周期）完成运算，因此不需要在代码中添加额外的时序机制。本程序可以替代之前的 mult_soft 程序。

17.5.2　UART 接口

使用 UART 接口，信息可以输入到 Windows 超级终端并显示，其灵活度和可视化程度比开关和 LED 要好很多。我们将其用作二次方计算程序的一个简单控制台。典型界面如图 17-8 所示。控制台会生成一个 SQ > 提示，用户可以输入小写字母 a、b、c 或 d 作为响应字符。a 和 b 字符用于为二次方计算程序的 a 和 b 输入数值。按键时，8bit 开关的数值会被读取并存储于 RAM 相应的地址空间中。c 字符用于复位数据 RAM，并对程序进行初始化，其功能和 c 按钮是一样的。d 字符用于数据 RAM 显示，RAM 中 64 字节的数据会在显示器上显示。这使我们可以观测二次方计算程序数值变化以及 7 段 LED 的数值。输入其他任何一个字符都会返回一个错误信息。

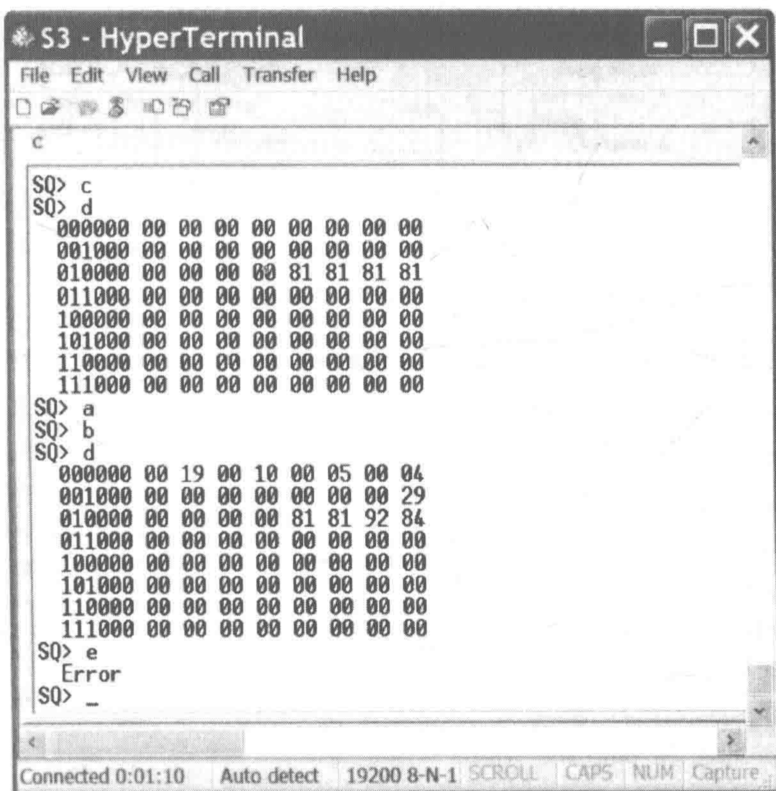


图 17-8　典型控制台界面

这里，我们可以使用 8.4 节中讨论的 UART 模块。由于发送和接收 FIFObuffer 提供了缓存和标识机制，所以无需额外添加任何电路。我们仅需要扩充译码和多路选择电路去包含额外的 I/O 端口。UART 接口的方框图在图 17-9 中进

行概要显示，其中其他的 I/O 端口被忽略以防止混淆。PicoBlaze 的输出端口 out-port 连接到 UART 的 a-data 端。解码使能信号连接到 wr-uart，当其有效时数据会被写入 UART 发送 FIFO。同样地，UART 的 r-data 连接到 PicoBlaze 的输入多路选择电路，解码后的复位信号被连接到 rd-uart。UART 接收 FIFO 端口在输入指令中指定，接收 FIFO 输出连接到 PicoBlaze 的输入端口 in-port，解码后的单时钟周期移出信号将一个字从接收 FIFO 中移出。UART 接口同样需要将 rx-empty 和 tx-full 两个状态信号连接到 PicoBlaze 输入多路选择电路。汇编程序需要在读写 UART 的 FIFO 前对其状态进行判读。因为信号仅有 2bit 位宽，我们可以将它和之前的 s 和 c 按钮整合成一组然后连接到同一个输入端口。

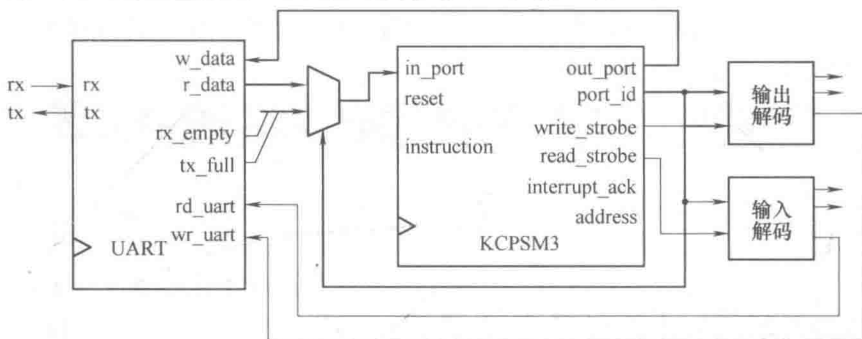


图 17-9　UART I/O 接口

17.5.3　汇编代码开发

由于之前汇编代码是以模块的方式进行开发，我们可以通过在汇编代码中增添子程序 prco-uart 的方式来实现 UART 传输的功能。主程序变为

```
    call init;initialization
forever:
    ;main loop body
    call proc_btn;check & process buttons
    call proc_uart;check & process uart rx
    call square;calculate square
    call load_led_pttn;store led patterns to ram
    call disp_led;output led pattern
    jump forever
```

由于控制台操作较为复杂，proc_uart 程序也会比较复杂。其伪代码如下：

```
;   if (no character in UART receiving FIFO) then
;       return
```

```
; input characters from FIFO
; if (characters is a) then
;    input switch value
;    store it to data ram
;    display prompt
;    return
; if (characters is b) then
;    input switch value
;    store it to data ram
;    display prompt
;    return
; if (characters is c) then
;    perform initialization
;    return
; if (characters is d) then
;    dump data ram
;    return
; display error message
;    return
```

我们遵循模块化开发的原则，可以进一步将本程序细化为几个简单的子程序。一个重要的底层子程序是 tx-one-byte 子程序。其功能为通过 UART 端口发送1字节数据，代码如下：

```
;input port definitions
constant rd_flag_port,00
; 4 flags (xxxxxtrsc):
;    t:uart tx full, r: uart rx not empty
;    s: s button flag, c: c button flag
;output port definitions
constant uart_tx_port,04;uart receiver port
;register alias
namereg sd,tx_data;data to be tx by uart
...
tx_one_byte:
   input s6,rd_flag_port
   test s6,08;check uart_tx_full
```

```
jump nz,tx_one_byte;yes, keep on waiting
output tx_data,uart_tx_port;no, write to uart tx fifo
return
```

因为 PicoBlaze 处理速度远远高于 UART 的发送速度，我们必须防止 buffer 上溢。本子程序会持续判读发送 FIFO buffer 的状态，只有在 buffer 不满时才向其写入数据。

显示数据 RAM 需要的工作量最大，将数据 RAM 的地址和内容显示为一个 8 行 8 列的表格首先列出字节的地址，接着以十六进制的方式给出 8 字节数据，如下：

```
001000  00  0F  00  09  00  04  00  03
010000  00  00  FF  1D  00  00  00  19
111000  00  00  00  00  00  FF  FF  FF
```

本程序包括 3 个主要的子程序：dispram-addr，用于发送 ASCII 码以二进制方式显示 5bit 基地址；disp_ram_data，用于发送 ASCII 码显示 8 字节数据；hex_to_ascii，将一个十六进制数转换为相应的 ASCII 码。

完整的代码在示例 17.3 中给出，并包括详细的注释来解释子程序的操作。示例 17.1 中未更改的子程序会被忽略。

示例 17.3　包含一个 UART 控制台的二次方计算程序

```
;==============================================
;square circuit with UART and multiplier interface
;==============================================
;program operation:
;   -read a and b from switch
;   -calculate a * a + b * b
;   -display data on HyperTerminal and 7-seg led
;==============================================
;data constants
;==============================================
;selected ASCII codes
constant ASCII_0,30
constant ASCII_1,31
constant ASCII_2,32
constant ASCII_3,33
constant ASCII_a,61
```

```
constant ASCII_b,62
constant ASCII_c,63
constant ASCII_d,64
constant ASCII_o,6F
constant ASCII_r,72
constant ASCII_E,45
constant ASCII_S,53
constant ASCII_Q,51
constant ASCII_D_U,44;uppercase D
constant ASCII_GT,3E; >
constant ASCII_SP,20;space
constant ASCII_CR,0D;carriage return
constant ASCII_LF,0A;line feed
;==============================================================
;data RAM address alias
;==============================================================
constant a_lsb,00
constant b_lsb,02
constant aa_lsb,04
constant aa_msb. 05
constant bb_lsb,06
constant bb_msb. 07
constant aabb_lsb,08
constant aabb_msb. 09
constant aabb_cout,0A
constant led0,10
constant led1,11
constant led2,12
constant led3,13
;==============================================================
;register alias
;==============================================================
;commonly used local variables
namereg s0,data;reg for temporary data
namereg s1,addr;reg for temporary mem & i/o port addr
```

```
namereg s2,I;general-purpose loop index
;global variables
namereg sc,switch_a_b;ram offset for current switch input
namereg sd,tx_data;data to be tx by uart
;==============================================================
;port alias
;==============================================================
;------------------------------input                                      port
definitions------------------------------------------------------
constant rd_flag_port,00
;4 flags (xxxxxtrsc):
;   t:uart tx full
;   r:uart rx not empty
;   s: s button flag
;   c: c button flag
constant sw_port,01;8-bit switchs
constant uart_rx_port,02;uart receiver port
constant mult_prod0_port,03;multiplication product 8 LSBs
constant mult_prod1_port,04;multiplication product 8 MSBs
;------------------------------output                                     port
definitions-----------------------------------------------------
constant sseg0_port,00;7-seg led 0
constant sseg1_port,01;7-seg led 1
constant sseg2_port,02;7-seg led 2
constant sseg3_port,03;7-seg led 3
constant uart_tx_port,04;uart receiver port
constant mult_src0_port,05;multiplier operand 0
constant mult_src1_port,06;multiplier operand 1
;==============================================================
;main program
;==============================================================
;calling hierarchy:
;
;main
;   -init
```

```
;      -tx_prompt
;         tx_one_byte
;   -proc_btn
;      -init
;   -proc_uart
;      -tx_prompt
;      -init
;      -proc_uart_err
;         -tx_one_byte
;      -dump_mem
;         -tx_prompt
;         -disp_ram_addr
;            -tx_one_byte
;            -get_upper_nibble
;            -get_lower_nibble
;            -hex_to_ascii
;   -square
;      -mult_hard
;   -load_led_pttn
;      -get_lower_nibble
;      -get_upper_nibble
;      -hex_to_led
;   -disp_led
;
;============================================================
  call init;initialization
forever:
  ;main loop body
  call proc_btn;check & process buttons
  call proc_uart;check & process uart rx
  call square;calculate square
  call load_led_pttn;store led patterns to ram
  call disp_led;output led pattern
  jump forever
;============================================================
```

```
;routine: init
;   function: perform initialization, clear register/ram
;   output register:
;      switch_a_b: cleared to 0
;   temp register: data,i
;=================================================
init:
;clear memory
  load I,40;unitize loop index to 64
  load data,00
clr_mem_loop:
  store data,(i)
  sub I,01;dec loop index
  jump nz,clr_mem_loop;repeat until i=0
  ;clear register
  load switch_a_b,00
  call tx_prompt
  return
;=================================================
;routine: proc_uart
;   function: read uart input char:
;      a or b: read a or b from switch;
;      c: clear; d: dump/display data ram other: error
;   input reg: s3(input port flag)
;   temp register used: data
;      s4: store received uart char or 00(no uart input)
;=================================================
proc_uart:
  test s3,04;check uart rx status
  jump z,uart_rx_done;go to done if rx empty
  ;process received char
  input s4,uart_rx_port;get char
  ;check if received char is a
  compare s4,ASCII_a;check ASCII a
  jump nz,chk_ascii_b;no,check next
```

```
  input data,sw_port;get switch
  store data,a_lsb; write a to data ram
  call tx_prompt;new prompt line
  jump uart_rx_done
chk_ascii_b:
  ;check if received char is b
  compare s4,ASCII_b;check ASCII b
  jump nz,chk_ascii_c;no,check next
  input data,sw_port;get switch
  store data,b_lsb;write b to data ram
  call tx_prompt;new prompt line
  jump uart_rx_done
chk_ascii_c:
  ;check if received char is c
  compare s4,ASCII_c;check ASCII c
  jump nz,chk_ascii_d;no check next
  call init;clear
  jump uart_rx_done
chk_ascii_d:
  ;check if received char is d
  compare s4,ASCII_d;check ASCII d
  jump nz,ascii_undefined
  call dump_mem;dump/display ram
  jump uart_rx_done
ascii_undefined
  ;undefined char
  call proc_uart_error
uart_rx_done:
  return
;===========================================================
;routine: proc_uart_error
;  function: display"Error" for unknown uart char
;===========================================================
proc_uart_error:
  load tx_data,ASCII_LF
```

```
  call tx_one_byte;transmit LF
  load tx_data,ASCII_CR
  call tx_one_byte;transmit CR
  load tx_data,ASCII_SP
  call tx_one_byte;transmit SP
  call tx_one_byte;transmit SP
  load tx_data,ASCII_E
  call tx_one_byte;transmit E
  load tx_data,ASCII_r
  call tx_one_byte;transmit r
  load tx_data,ASCII_r
  call tx_one_byte;transmit r
  load tx_data,ASCII_o
  call tx_one_byte;transmit o
  load tx_data,ASCII_r
  call tx_one_byte;transmit r
  call tx_prompt
  return
;=========================================================
;routine: dump_mem
;  function: when d received, dump 64 bytes of ram as
;     001000 XX XX XX XX XX XX XX XX
;     010000 XX XX XX XX XX XX XX XX
;…
;     111000 XX XX XX XX XX XX XX XX
;  temp register used:
;     s3: as outer loop index
;     s4: ram base address
;=========================================================
dump_mem:
  load s3,00;addr used as loop index
dump_loop:
  ;loop body
load s4,s3;get ram base addr(xxx000)
s10 s4
```

```
s10 s4
s10 s4
call disp_ram_addr
call disp_ram_data
add s3,01;inc loop index
compare s3, 08
jump nx,dump_loop;loop not reach 8 yet
call tx_prompt;new prompt
return
;================================================================
;routine: tx_prompt
;   function: generate prompt "SQ >"
;   temp register: tx_data
;================================================================
tx_prompt:
  load tx_data,ASCII_LF
  call tx_one_byte;transmit LF
  load tx_data,ASCII_CR
  call tx_one_byte;transmit CR
  load tx_data,ASCII_S
  call tx_one_byte;transmit S
  load tx_data,ASCII_Q
  call tx_one_byte;transmit Q
  load tx_data,ASCII_GT
  call tx_one_byte;transmit >
  load tx_data,ASCII_SP
  call tx_ont_byte;transmit SP
  return
;================================================================
;routine:disp_ram_addr
;   function: display 6-bit ram addr
;   bbb000
;   input register:
;      s4: base address
;   temp register:
```

```
;i,s7: 1-bit mask
;==================================================
disp_ram_addr:
   ;new line
   load tx_data,ASCII_LF
   call tx_one_byte;transmit LF
   load tx_data,ASCII_CR
   call tx_one_type;transmit CR
   load tx_data,ASCII_SP
   call tx_one_byte;transmit SP
   call tx_one_byte;transmit SP
   ;initialize the loop index and mask
   load i,06;addr used as loop index
   load s7,20;set mask to 0010_0000
tx_loop:
   ;loop body
   load tx_data,ASCII_1;load default ASCII 1
   test s7,s4;check the bit
   jump nz,tx_01;the bit is 1
   load tx_data,ASCII_0;the bit is 0,load ASCII 0
tx_01:
   call tx_one_byte;transmit the ASCII 1 or 0
   ;update loop index and mask
   sr0 s7;shift mask bit
   sub i,01;dec loop index
   jump nz,tx_loop;loop not reach 0 yet
   ;done with loop,send ASCII space
   load tx_data,ASCII_SP;load ASCII SP
   call tx_one_byte;transmit SP
   return
;==================================================
;routine:disp_ram_data
;  function: 8-byte data in form of
;     00 11 22 33 44 55 66 77 88
;  input register:
```

```
;     s4: ram base address(xxx000)
;  temp register:i,addr,data
;===================================================
disp_ram_data:
   ;initialize the loop index and mask
   load I,08;addr used as loop index
d_ram_loop:
   ;loop body
   load addr,s4
   add addr,i
   sub addr,01;calculate addr offset
   ;send upper nibble
   fetch data,(addr)
   call get_upper_nibble
   call hex_to_ascii;convert to ascii
   load tx_data,data
   call tx_one_byte
   ;send lower nibble
   fetch data,(addr)
   call get_lower_nibble
   call hex_to_ascii;convert to ascii
   load tx_data,data
   call tx_one_type
   ;send a space
   load tx_data,ASCII_SP;
   call tx_one_byte;transmit SP
   sub i,01;dec loop index
   jump nz,d_ram_loop;loop not reach 0 yet
   return
;===================================================
;routine: hex_to_ascii
;  function: convert a hex number toascii code
;            add 30 for 0-9, add 37 for A-F
;  input register: data
;===================================================
```

```
hex_to_ascii:
  compare data,0a
  jump c,add_30;0 to 9, offset 30
  add data,07;a to f,extra offset 07
add_30:
  add data,30
  return
;==========================================================
;routine: tx_one_byte
;  function: wait until uart tx fifo not full;
;            then write a byte to fifo
;  input register: tx_data
;  temp register:
;     s6: read port flag
;==========================================================
tx_one_byte:
  input s6,rd_flag_port
  test s6,08;check uart_tx_full
  jump nz,tx_one_byte;yes, keep on waiting
  output tx_data,uart_tx_port;no,write to uart tx fifo
  return
;==========================================================
;routine: square
;  function: calculate a * a + b * b
;     data/result stored in ram started w/SQ_BAST_ADDR
;  temp register: s3,s4,s5,s6,data
;==========================================================
square:
   ;calculate a * a
  fetch s3,a_lsb;load a
  fetch s4,a_lsb;load a
  call mult_hard;calculate a * a
  store s6,aa_lsb;store lower byte of a * a
  store s5,aa_msb;store upper byte of a * a
   ;calculate b * b
```

```
  fetch s3,b_lsb;load b
  fetch s4,b_lsb;load b
  call mult_hard;calculate b * b
  store s6,bb_lsb;store lower byte of b * b
  store s5,bb_msb;store upper byte of b * b
  ;calculate a * a + b * b
  fetch data,aa_lsb;get lower byte of a * a
  add data,s6;add lower byte of a * a + b * b
  store data,aabb_lsb;store lower byte of a * a + b * b
  fetch data,aa_msb;get upper byte of a * a
  addcy data,s5;add upper byte of a * a + b * b
  store data,aabb_msb;store upper byte of a * a + b * b
  load data,00;get carry from previous
  store data,aabb_cout;store carry of a * a + b * b
  return
;=============================================================
;routine:mult_hard
;  function: 8-bit unsigned multiplication using
;            external combinational multiplier;
;  input register:
;     s3: multiplicand
;     s4: multiplier
;  output register:
;     s5: upper byte of product
;     s6: lower byte of product
;  temp register:
;=============================================================
mult_hard:
  output s3,mult_src0_port
  output s4,mult_src1_port
  input s5,mult_prod1_port
  input s6,mult_prod0_port
  return
;=============================================================
;The following are the same as the previous listings:
```

```
;  proc_btn, load_led_pttn, disp_led
;  hex_to_led, get_lower_nibble, get_upper_nibble
;…
;==========================================================
```

17.5.4 HDL 代码开发

新的二次方电路向 I/O 接口添加了一个 UART 和一个组合乘法器。前者即为 8.4 节中讨论的模块，后者可以通过 HDL 的“ * ”运算符导出。示例 17.2 中 HDL 代码的解码和多路选择部分可以被扩充来包含两个新的外设。完整的 HDL 代码在示例 17.4 中显示。详细的 I/O 端口地址分配可以在之前一个章节（17.3 节）中找到。

示例 17.4 包括 UART 控制台和乘法器接口的 PicoBlaze

```
module pico_uart
  (
    input wire clk,reset,
    input wire[7:0] sw,
    input wire rx,
    input wire[1:0]btn,
    output wire tx,
    output wire[3:0] an,
    output wire[7:0]sseg
  );
  // 信号声明
  //KCPSM3/ROM 信号
  wire[9:0] address;
  wire[17:0] instruction;
  wire[7:0] port_id,out_port;
  reg[7:0] in_port;
  wire   write_strobe,read_strobe;
  //I/O 端口信号
  // 输出使能
  reg[6:0] en_d;
  //4 位7 段LED 显示
```

```
reg[7:0] ds3_reg,ds2_reg,ds1_reg,ds0_reg;
//两个按钮
reg btnc_flag_reg,btns_flag_reg;
wire btnc_flag_next,btns_flag_next;
wire set_btnc_flag,set_btns_flag,clr_btn_flag;
//uart
wire[7:0] rx_char;
wire rd_uart,rx_not_empty,rx_empty;
wire wr_uart,tx_full;
//乘数
reg[7:0] m_src0_reg,m_src1_reg;
wire[15:0] prod;
//实体
//=====================================================
//I/O 模块
//=====================================================
disp_mux disp_unit
   (.clk(clk),.reset(reset),
    .in3(ds3_reg),.in2(ds2_reg),.in1(ds1_reg),
    .in0(ds0_reg),.an(an),.sseg(sseg));
debounce btnc_unit
   (.clk(clk),.reset(reset),.sw(btn[0]),
    .db_level(),.db_tick(set_btnc_flag));
debounce btns_unit
   (.clk(clk),.reset(reset),.sw(btn[1]),
    .db_level(),.db_tick(set_btns_flag));
uart uart_unit
   (.clk(clk),.reset(reset),.rd_uart(rd_uart),
    .wr_uart(wr_uart),.rx(rx),
    .w_data(out_port),.tx_full(tx_full),
    .rx_empty(rx_empty),.r_data(rx_char),.tx(tx));
//组合乘数
assign prod = m_src0_reg * m_src1_reg;
//=====================================================
//KCPSM 和ROM 实例
```

```
// ==============================================
  kcpsm3 proc_unit
    (.clk(clk),.reset(1'b0),.address(address),
      .instruction(instruction),.port_id(port_id),
      .write_strobe(write_strobe),.out_port(out_port),
      .read_strobe(read_strobe),.in_port(in_port),
      .interrupt(1'b0),.interrupt_ack());
  uart_rom rom_unit
    (.clk(clk),.address(address),
      .instruction(instruction));
// ==============================================
  // 输出接口
// ==============================================
  // 输出端口 id:
  //  0x00 : ds0
  //  0x01 : ds1
  //  0x02 : ds2
  //  0x03 : ds3
  //  0x04 :uart_tx_fifo
  //  0x05 : m_src0
  //  0x06 : m_src1
// ==============================================
  // 寄存器
  always @ (posedge clk)
    begin
      if(en_d[0])
        ds0_reg <= out_port;
      if(en_d[1])
        ds1_reg <= out_port;
      if(en_d[2])
        ds2_reg <= out_port;
      if(en_d[3])
        ds3_reg <= out_port;
      if(en_d[5])
        m_src0_reg <= out_port;
```

```
      if(en_d[6])
         m_src1_reg <= out_port;
   end
// 解码电路使能信号
always @ *
   if(write_strobe)
      case(port_id[2:0])
         3'b000: en_d = 7'b0000001;
         3'b001: en_d = 7'b0000010;
         3'b010: en_d = 7'b0000100;
         3'b011: en_d = 7'b0001000;
         3'b100: en_d = 7'b0010000;
         3'b101: en_d = 7'b0100000;
         default: en_d = 7'b1000000;
      endcase
   else
      en_d = 7'd0000000;
assign wr_uart = en_d[4];
// ==========================================================
// 输入接口
// ==========================================================
// 输入端口 id
//  0x00 : flag
//  0x01 : switch
//  0x02 :uart_rx_fifo
//  0x03 : prod lower byte
//  0x04 : prod upper byte
// ==========================================================
// 输入寄存器
always @ (posedge clk)
   begin
      btnc_flag_reg <= btnc_flag_next;
      btns_flag_reg <= btns_flag_next;
   end
assign btnc_flag_next = (set_btnc_flag) ? 1'b1:
```

```
                        (clr_btn_flag) ? 1'b0:
                        btnc_flag_reg;
assign btns_flag_next = (set_btns_flag) ? 1'b1:
                        (clr_btn_flag) ? 1'b0:
                        btns_flag_reg;
// 信号清晰的解码电路
assign clr_btn_flag = read_strobe && (port_id[2:0] = =3'b000);
assign rd_uart = read_strobe && (port_id[2:0] = =3'b010);
// 输入多路复用
assign rx_not_empty = ~rx_empty;
always @ *
  case(port_id[2:0])
    3'b000: in_port = {1'b0,tx_full,rx_not_empty,
                    btns_flag_reg,btnc_flag_reg};
    3'b001: in_port = sw;
    3'b010:in_port = rx_char;
    3'b011:in_port = prod[7:0];
    default:in_port = prod[15:8];
  endcase
endmodule
```

17.6 文献备注

本章的文献备注信息同第 15 章相似。下载的 kcpsm 文件中包含一个完整的 UART 例子和一个定时器设计例子。Xilinx 网站上有关于“PicoBlaze 讨论”和“PicoBlaze 用户资源”的专栏，其中有更丰富的 PicoBlaze 例子。

17.7 实验

17.7.1 低频计数器 I

精确的低频计数器在 6.3.5 节中已经讨论过。我们可以将周期计数器、分频电路和二进制到十进制的转换电路视为 3 个 I/O 模块，并用 PicoBlaze 替代顶层状态机。设计 I/O 接口，进行汇编代码和 HDL 代码设计，编译并综合电路，验

证其功能。

17.7.2　低频计数器 II

我们可以利用软件子程序去替代 17.7.1 节频率计数器中分频电路和二进制到十进制转换电路的硬件。重新设计 I/O 接口，进行汇编代码和 HDL 代码设计，编译并综合电路，验证其功能。

17.7.3　自适应低频计数器

实验 6.5.5 中讨论了一个自适应低频计数器。我们可以利用 PicoBlaze 去实现所有非时间敏感功能。结合 PicoBlaze 重新设计电路，最小化外围硬件。进行汇编代码和 HDL 代码设计，编译并综合电路，验证其功能。

17.7.4　利用软件定时器替代基础反应定时器

实验 6.5.6 讨论过反应定时器，现在我们可以利用 PicoBlaze 重新设计这个电路。首先需要记录过去的时间间隔，这个可以由一个软件计数器子程序来实现。原型开发板上的时钟为 50MHz，每一个指令需要耗费两个时钟周期。建立有一个计数循环来记录指令执行的次数，并据此得出时间间隔。因为间隔至少是 ms 级，因此需要使用多个寄存器。设计 I/O 接口，进行汇编代码和 HDL 代码设计，编译并综合电路，进行功能验证。

17.7.5　包含硬件定时器的反应定时器

我们可以利用一个定制的硬件定时器来重做 17.7.4 节中的实验。定时器应被视为一个 I/O 外设。PicoBlaze 可以通过输出一个命令去清空、起动或暂停定时器，并可以获取定时器的当前值。设计 I/O 接口，进行汇编代码和 HDL 代码设计，编译并综合电路，并进行功能的验证。

17.7.6　增强型反应定时器

一个改进的反应定时器可以记录最近的四次响应时间、最快的响应时间，并能在 Windows 的超级终端上显示以上这些数据。我们可以设计一个类似于 17.5 节的控制台，这里需要有 3 个命令：

- c：清空所有的数据。
- f：显示最快的响应。
- r：显示最近的四次响应时间。
- 其他的字符：显示“error”。

扩展 17.7.4 节和 17.7.5 节中的设计来实现该功能。设计 I/O 接口，进行汇

编代码和 HDL 代码设计，编译并综合电路，进行功能验证。

17.7.7 小屏幕鼠标跟踪电路

13.7.10 节讨论过小屏幕鼠标跟踪电路的设计。我们可以利用 PicoBlaze 去监控鼠标的动作并更新相关的影像内存。设计 I/O 接口，进行汇编代码和 HDL 代码设计，编译并综合电路，进行功能验证。

17.7.8 全屏幕鼠标跟踪电路

13.7.11 节中讨论过一个全屏幕数遍跟踪电路，这里我们可以用 PicoBlaze 去监控鼠标的动作并更新相关的影像内存。设计 I/O 接口，进行汇编代码和 HDL 代码设计，编译并综合电路，进行功能验证。

17.7.9 增强型跑马灯字幕

14.6.1 节中讨论过 VGA 跑马灯字幕的电路设计。我们可以通过使用键盘输入方向信息的方式增强电路的性能。假定信息 buffer 有 20 个字符长，其字符更新依据先入先出的方式。使用 PicoBlaze 重新设计电路。设计 I/O 接口，进行汇编代码和 HDL 代码设计，编译并综合电路，进行功能验证。

17.7.10 乒乓游戏

14.4 节中乒乓游戏的部分功能可以通过 PicoBlaze 来实现：

- 顶层控制状态机。
- 顶层 2s 定时器和两位数十进制计数器。
- 示例 13.5 用于更新弹板位置，球的位置以及球的速率。

更改之前的电路，设计 I/O 接口，进行汇编代码和 HDL 代码设计，编译并综合电路，进行功能验证。

17.7.11 文本编辑器

14.6.5 节的实验讨论了一个 UART 终端，我们可以用 PicoBlaze 去通过 UART 获取数据和命令，从而更新存储块。设计 I/O 接口，进行汇编代码和 HDL 代码设计，编译并综合电路，进行功能验证。

第 18 章　PicoBlaze 中断接口

18.1　简介

在程序正常运行的过程中，微处理器负责通过 I/O 设备（例如检查状态信号），以及决定下一步正确的运行进程。I/O 设备只是被动地等待该它运行的时刻。中断是一种允许 I/O 设备开始运转的信号机制。就像这名字一样，中断是一个常规的可执行的程序，用来开启 I/O 设备的服务行程。对于一个微处理器来说，中断常常用于一些对时间因素要求很高的外围设备操作，遇到危急时刻，中断可以立即执行。PicoBlaze 微处理器就具有提供简单的中断操作的性能。在本章中，我们要检查这个 PicoBlaze 的中断机制，以及举例说明软件和接口技术的发展。

18.2　PicoBlaze 里的中断操作

中断操作在硬件和软件里的重要性是一致的。当一个外部设备需要通过中断来运行，它就声明 PicoBlaze 中断信号。如果这个中断信号是可执行的，PicoBlaze 就执行当前指令，使得中断命令信号接收中断请求，然后就自动执行地址 3FF 寄存器的指令。当这条指令被执行了，当前的程序进度被保存在堆栈里，地址 3FF 也被导入程序进度里。注意，这个 3FF 地址是最后一个指令寄存器的地址，也是开始指向中断服务程序的地址。它通常包含一个跳变指令，引导服务程序的主体。当一个反馈指令执行使得进程返回到中断处，继续执行之前被中断的进程的时候，中断服务在此时就结束了。

18.2.1　软件处理

4 个指令是和中断相关联的，这在 15.5.9 节已讨论过。中断使能以及禁止中断指令可以许可或禁止中断请求。这两种相反的中断指令，执行之后都会返回到中断点。

图 18-1 是一个典型的中断服务程序流程图。它通常包含如下部分：

● 一个初始化的中断使能指令，用来允许中断服务，这个是必须的，因为默认的中断请求是处于无效状态；

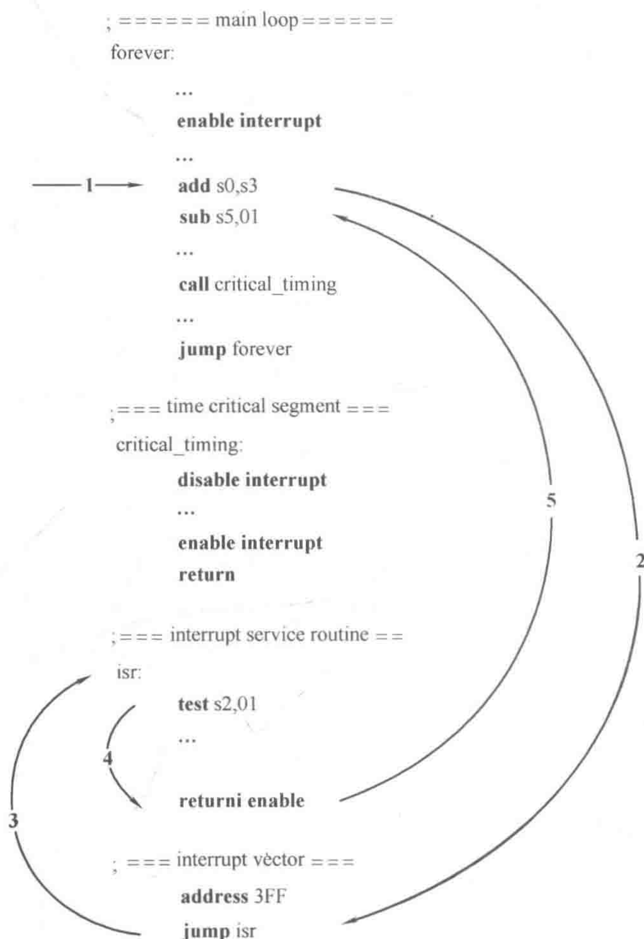


图 18-1　中断的流程图

●一个跃升指令在指令寄存器的末端（例如 3FF）：触发中断服务程序；

●中断服务程序：这个代码实际上执行的是被请求的服务，这个程序应该被中止，用一个回馈指令。

中断事件的代表性流程图如图 18-1 所示。我们假定这个外部的 I/O 接口断定中断信号在增加的 S0、S3 指令的中间。PicoBlaze 按顺序执行以下各步骤：

1）完成当前执行；

2）保留程序数的进程，清理中断标记 i 为 0，实现 0 保护和达到标记点，从 3FF 里读取程序数；

3）在地址 3FF 里执行跃升中断服务程序指令；

4）执行服务程序；

5）执行反馈指令，复原被保存的程序进程和标记点；

6）继续执行中断程序，执行 S5、01 指令。

18.2.2　时序图

之前的中断事件的详细时序图如图 18-2 所示。基本的顺序是：

- 在 t1 时刻，外部的中断接口生成中断信号，PicoBlaze 继续正常的操作，完成执行当前增加指令 s0，s3；
- 在 t2 时刻，PicoBlaze 认可中断，使得下一个指令失效（S5，01），间接执行 3FF 的指令；
- 在 t3 时刻，PicoBlaze 发出中断命令正确应答信号，它也保存了 s5，01 指令的地址，实现 0 保护和传送标记，清理中断标记为 0；
- 在 t4 时刻，PicoBlaze 加载执行 3FF 地址里的指令，跳出中断服务程序，外部中断接口依次对中断命令正确应答信号做出反馈，回收中断信号；
- 在 t5 时刻，PicoBlaze 开始中断服务程序。

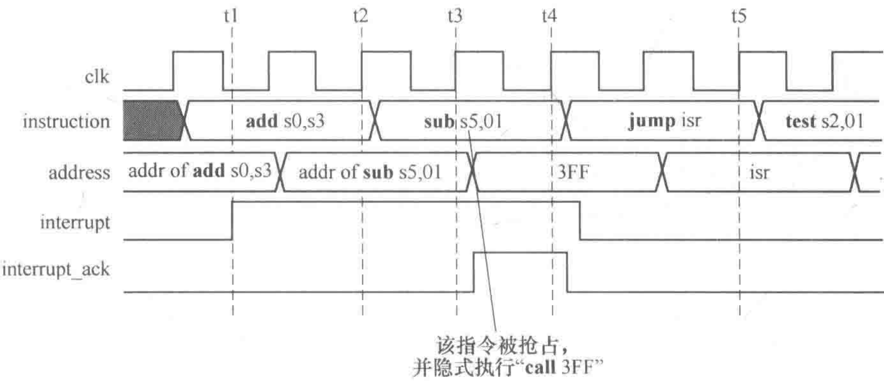


图 18-2　一个中断时间的时序图

注意，它需要 5 个时钟周期，从中断信号发出时刻起，到第一个中断服务指令被执行为止。

18.3　外部接口

中断请求的本质类似于在 17.3.2 节里论述的信号通路端口。当请求被接受的时候，它必须被清理掉，防止相同的请求被多次执行。在 8.2.4 节论述的标记点 FF 可以被用来达到这个目的。

18.3.1　中断请求信号

如果在 PicoBlaze 系统里只有一个外围的 I/O 接口的话，那就可以产生一个

中断请求，我们只需要一个标记触发器在中断接口周期，如图 18-3 所示。当需要这个服务的时候，外部 I/O 接口电路发出一个时钟周期的 int 请求信号，把特征位触发器设置为 1，激活 PicoBlaze 的中断输入。如果 PicoBlaze 的中断是被许可的，发出一个时钟周期的中断命令正确信号，来接收这个请求，把特征位触发器清零。

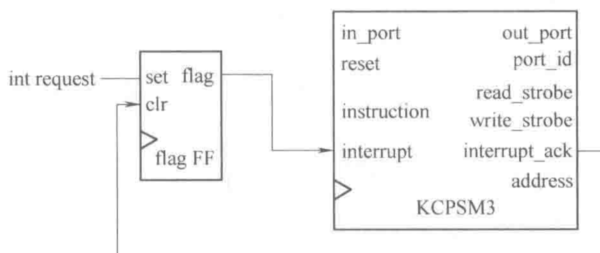


图 18-3　一个中断请求信号的接口

18.3.2　多重中断请求

处理一个有两个中断请求以上的 PicoBlaze 系统则更棘手。PicoBlaze 的微处理器必须确定究竟是哪一个外部设备发出的中断请求以及在完成中断请求后清除掉相应的标记触发器。这需要外部硬件接口以及中断服务程序的协调操作。

中断接口的两个请求，如图 18-4 所示。这两个独立的请求，标准请求 0 和标准请求 1，和两个标记触发器相联系，触发器是输出信号被传递给产生最终中断请求信号的门器件。除此之外，这两个信号也发送到多路输入器。如果至少有一个请求等待执行，PicoBlaze 的中断信号就发出了。当 PicoBlaze 察觉到这个请求，它不知道到底是哪个外围设备，或者是几个外围设备一起发出的这个请求。这个中断服务程序必须首先输入这两个请求信号，然后检查它们的确切含义，根据设定好的优先权，然后执行相应的服务程序。

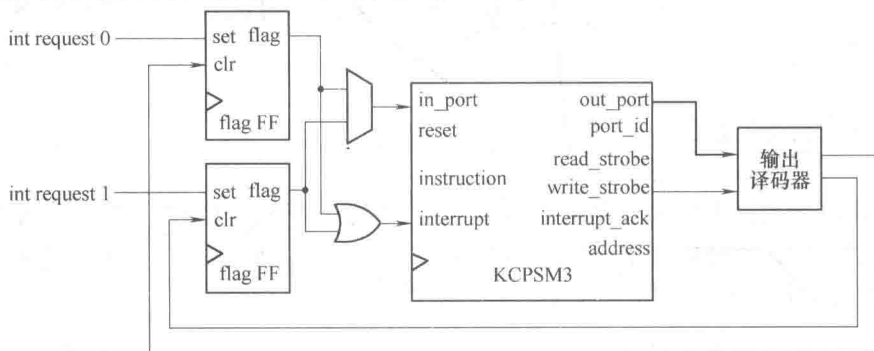


图 18-4　两个请求的中断接口

除此之外，PicoBlaze 也需要清理相应的标记触发器，这个中断正确指令信号不能被用来达到这个目的，因为它不知道到底哪一个外围设备的请求是被接收的，当中断正确指令信号发出后。我们可以这样，我们需要用一个特殊的输出解码电路产生一个清除记号。这每一个标记触发器的 clr 信号被指派到唯一对应的端口 id。在中断服务程序里，在决定了哪一个中断请求被接收了之后，我们增加一个输出指令。这个指令不能输出任何数据。它只是用来产生一个信号周期来清除相应的标记触发器。

为了减少软件顶层构架，加快响应速度，我们可以设计一个中断控制器，使得这个进程变得更容易一些。这个方法将在 18.7.5 节以试验形式讨论。

18.4 软件发展描述

18.4.1 中断作为一个可选择的计划方案

回看一款基于微处理器的应用常常有一套简单的表决项目结构：

```
call initialization_routine
forever:
call taskl_routine;
call task2_routine;
...
call taskn-routine;
jump forever;
```

有些任务也许包括 I/O 操作。在执行期间，微处理器按顺序检查 I/O 状态，采取相应的操作。这种程序结构实现了环状进度表，每一个任务都按顺序等待被执行。这种配置可以完全的工作，如果循环间隔短到每一个 I/O 请求都可以被检查和在合适的时间期限内处理。在一些应用里，也许存在一个或两个时间周期的 I/O 请求，这就需要更直接的关注。中断机制提供一种方式，打断初始的计划安排，给一个确定的后续任务更高的优先权。

既然一个中断可能发生在任何时间，这个原始闭环必须考虑到中断的频率以及被需求的每个中断请求的服务时间。当有多个相关的中断请求或是服务程序的时候，这可能比较复杂。

18.4.2 中断服务程序的发展

中断服务程序就好比一个子程序。它使得正常的程序延缓，执行独立的任务，然后恢复先前执行的程序。然而和子程序不同的是，中断可能发生在任何时

间。为了恢复先前执行的程序，这个服务程序必须保存 PicoBlaze 处理器的当前状态（也叫做当前背景）。换句话说，这个服务程序必须保存所有该服务程序中用于计算的寄存器，然后恢复它们，在返回正常执行程序之前。这个进程被称为背景交换。

既然 PicoBlaze 是一个简单的 8 位微处理器，这个背景交换的计划的硬件支持也很有限。我们应该用通用的表决方案，保持中断结构的简单化和易懂性。我们可以分配用于该服务的寄存器用来专门服务于中断服务程序，而不是担心背景交换。

18.5　设计用例

第 17 章用的方形回路，采用 7 段式的 LED 来显示输入的操作数和结果。我们用预定的 LED 多路技术模块达到显示的目的。这种模块设计在 4.5.1 节已经讨论过，它由一个很大的计数器产生一个很缓慢的使能脉冲，多回路则为发送模式。

为了节省硬件，我们可以在软件里贯彻这种功能，让 PicoBlaze 控制这个 4 位的使能信号，8 位的 LED 信号，4 位直接显示。为了产生一个真实的连续的样本，这个使能脉冲和 LED 端口必须定期更新，这在图 4-6 里有所表示。用纯软件保留时钟路径是可能的，但这种代码易错。我们用一个专门的计时器硬件和 PicoBlaze 的中断设施来完成这个任务。这种硬件需求和软件更改在下面章节里都有图示。

18.5.1　接口中断

时钟和接口中断的阻塞图表，也是新的输出块，如图 18-5 所示。计时器是一个 500-mod 的计时器，每隔 500 个时钟周期产生一个信号时钟周期。当这 50MHz 的时钟被用在这个计时器上的时候，这个 tick 的周期是 0.01ms。因为这有唯一的中断请求，我们用触发器标记接口中断，这在 18.3.1 节已讨论。这个 tick 设置触发器标记，刺激 PicoBlaze 的中断信号。

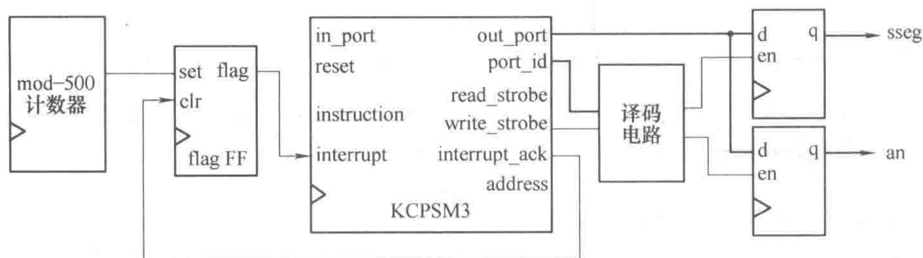


图 18-5　一个计时器的中断接口

18.5.2　中断服务程序的发展

为了保留消逝时钟的路径，PicoBlaze 就对时钟周期进行计数。如 18.4.2 节所讨论的，我们希望保持这个中断服务程序简单化，用两个专门的寄存器：count_msb 和 count_lsb 来完成这个任务。这两个寄存器是层叠的，作为一个 16 位的寄存器，增加每一个中断服务程序被响应的时间。它们可以计数到 0.6s 的精确度。这里叙述的中断代码片段如下：

```
namereg se, count_msb ; timer tick count 8 MSBs
namereg sf, count_lsb ; timer tick count 8 LSBs
. . .
;interrupt service routine
Int_service_routine:
add count_lsb, 01 ; inc 16-bit counter
addcy count_msb, 00
returni enable
;interrupt vector
address 3FF
jump int_service_routine
```

18.5.3　集成代码的发展

时序信息是可用的，因此我们可得到一个新的子程序，用 LED 来显示 mux_ out。在第 17 章中，这个程序代替显示 LED 的程序。需要两个新的输出缓冲器来存储 an 和 sseg 信号，这在 18.5 节有所述叙。这个子程序的主要任务就是存储这个采样数据，它可能是 1110，1101，1011，0111，同时，相应的七位 LED 显示器也模拟出周期性的记录表。就如 4.5.1 节所述，更新频率应该稳定在数百赫兹到数千赫兹之间。在我们的代码里面，我们更新这些寄存器，每隔 1024 个周期脉冲，大致相当于 10ms。我们也可以用一个寄存器 led_ pos 来跟踪记录当前显示排列，例如 4 个 LED 显示器中的一个。

为了使得新的中断特点能够融入示例 17.3，使之成为一体化结构，这个代码还应该做出如下修改：

- 增加新的端口，定义新的寄存器；
- 用 display_ mux-out 程序代替最初的 disp-led；
- 在内部的程序里增加使能中断指示，使得中断操作能够执行；
- 在内部程序里面初始化 led_ pos，count_ msb 以及 count_ lsb 寄存器；
- 增加中断服务程序。

示例 18.1 就是部分修改过的代码汇编。

示例 18.1　与中断接口方的程序

```
...
;register alias
namereg sb, led-pos ; led disp position(0, 1, 2 or 3)
namereg se, count-msb ; timer tick count 8 MSBs
5   namereg sf, count-lsb ; timer tick count 8 LSBs
...
;output port definitions
constant an_port , 00
constant sseg_port , 01
10  ...
;main program
call init ; initialization
forever:
;main loop body
15  call proc_btn ; chenk & process bottons
call square ; calculate square
call load_led_pttn ; atore led patterns to ram
call display-mux-out ; multiplex led patterns
jump forever
20
;// =====================================================
;routine: init
;// =====================================================
init:
5   enable interrupt
. . .
load led-pos, 00
load count-msb, 00
load count-lsb, 00
30  return
// =====================================================
;routine : display_mux_out
```

```
;funtikon : generate enable pulse & led pattern
35 ;    for 4-digit  7-segment led display
;input register :
count-msb, count_lsb : time count
led_pos : current led position
;output register:
40;    led_pos : updated led position
;tmp register : data , addr
// ====================================================
Display_mux_out :
compare count_msb , 02 ; count =00000100_00000000
45   jump c , mux-out-done
;clear time counter(count > 20)
load count_lsb, 00
load count_msb, 00
;update 7-segment led position
50   add led_pos, 01
compare led_pos, 04
jump nz , gen_an_signal
load led_pos, 00 ; led_pos wraps around
gen_an_signal:
55   ; generate 4 -bit anode enable signal
load data, OE ;xxxx - 1110
compare led_pos, 00
jump z, shift_an_0
compare led_pos, 01
60   jump z , shift_an_1
compare led_pos, 02
jump z,shift_an_2
sll data ; shift 1110 3 tmes
shift_an_2:
65   sll data ; shift 1110 2 tmes
shift_an_1:
sll data ; shift 1110 1 tmes
shift_an_0:
```

```
output data, an_port
70  : output 7-seg led pattern
load addr, led0
add addr, led-pos
fetch data, (addr)
output data, sseg-port
75  mux_out_done :
return
;routine : interrupt service routine
80  : function : increment 16_bit counter
;input register :
count-msb, count_lsb : timer count
:output register :
count-msb, count_lsb : incremented
// ============================================================
Int_service_routine:
add count_lsb, 01 ; inc 16-bit counter
addcy count_msb , 00
returni enable
90
// ============================================================
;interrupt vector
// ============================================================
address 3FF
95  jump int_service_routine
// ============================================================
:The following are the same as the previkous listings:
;proc_btn, load_led_pttn,
100; hex_to_led, get_lowre_nibble, get_upper_nibble
;square, mult_soft
, . . .
```

18.5.4 HDL 代码的发展

基于中断的 I/O 接口方形回路，包括 3 个部分。输入接口和 17.4 节所述是

相似的。输出接口是由一个译码电路和两个输出寄存器组成，一个是 an 信号，一个是 sseg 信号，如图 18-5 右侧所示。中断接口是由一个计时器和一个触发器标记组成的，如图 18-5 左侧所示。HDL 代码基本上遵循阻塞图表，如示例 18.2 所示。

示例 18.2　PicoBlaze 的基于方形回路的中断

```
module pico-int
(
input wire clk, reset,
input wire[ 7 : 0 ] sw,
input wire[ 1 : 0 ] btn,
output wire[3:0] an,
output wire[ 7 : 0 ] sseg
);
// 信号声明
//KCPSM3/ROM 信号
wire[ 9 : 0 ] address;
wire[ 17 : 0 ] instruction;
wire[ 7 : 0] port_id , out_port ;
reg[7:0] in_port ;
wire write_strobe , read_strobe ;
wire interrupt , interrupt_ack ;
//I/O 端口信号
// 输出使能
reg[ 1 : 0 1 en-d;
//4 位7 段LED 显示
reg[ 7 : 0 1 sseg-reg;
reg[3:01 an-reg;
// 两个按钮
reg btnc_f lag_reg , btns_f lag_reg ;
wire btnc_f lag_next , btns_f lag_next ;
wire set_btnc_flag, set_btns_flag, clr_btn_flag;
// 中断相关的信号
reg[8: 0] timer_reg;
wire[ 8 : 0 ] timer_next ;
```

```
wire ten_us_tick;
reg timer_f lag_reg ;
wire timer_f lag_next ;
// 实体
// ================================================
//I/O 模块
// ================================================
debounce btnc_unit
(. clk(clk), . reset(reset), . sw(btn[0]),
. db-level(),. db-tick(set-btnc-flag));
debounce btns-unit
( . clk(c1k) , . reset (reset ) , . sw (btn Ell ,
. db-level(),. db-tick(set-btns-flag));
// ================================================
//KCPSM 和ROM 实例
// ================================================
kcpsm3 proc_unit
(. clk(clk),. reset(1'b0),. address(address),
. instruction(instruction), . port_id(port_id),
. write_strobe(write_strobe), . out_port(out_port),
. read_strobe(read_strobe), . in_port(in_port),
. interrupt(interrupt),. interrupt_ack(interrupt_ack));
Int_rom rom_unit
( . clk(clk) , . address (address),
. instruction(instruction));
// ================================================
// 输出接口
// ================================================
// 输出端口 id :
//0-x-00 : an
//0 x 0 1 : s s g
// 寄存器
always @ (posedge clk)
begin
if(en_d[0])
```

```
an_reg <= out_port[3:0] ;
if (en_d[1])
            sseg_reg <= out_port ;
end
assign an = an_reg;
assign sseg = sseg_reg;
// 解码电路使能信号
always @ *
if(write_strobe)
case (port-id[0])
1'b0: en_d =2'bb01;
1'b1: en_d =2'b10;
      endcase
else
en_d =2'b00;
// ==========================================================
// 输入接口
// ==========================================================
// 输入端口 id
//0 x 0 0  :f l a g
//0 x 0 1  :s w i t c h
// 输入寄存器
always @ (posedge clk)
begin
btnc_flag_reg <= btnc_flag_next;
btns_flag_reg <= btns_flag_next;
      end
assign btnc_flag_next = (set_btnc_flag) ? 1'b1 :
(clr_btn_flag) ? 1'b0:
btnc-flag-reg;
assign btns-flag-next = (set-btns-flag) ? 1'b1 :
                  (clr-btn-flag) ? 1'b0 :
btns-flag-reg;
// 信号清晰的解码电路
assign clr_btn_f lag = read_strobe && (port_id[0] = =1'b0) ;
```

```
// 输入多路复用
always @ *
case (port_id[0]
1'b0: in_port = {6'b0, btns_flag_reg, btnc_flag_reg};
1'b1: in_port = sw;
endcase
// ================================================================
// 中断接口
// ================================================================
//10 μs 计数器
always @ (posedge clk)
timer_reg <= timer_next ;
assign ten_us_tick = (timer_reg = =499) ;
assign timer_next = ten_us_tick ? 0 : timer_reg + 1;
//10 μs t i c k f l a g
always @ (posedge clk)
timer_flag_reg <= timer_flag_next;
assign timer_flag_next = (ten_us_tick) ? 1'b1 :
(interrupt_ack) ? 1'b0:
timer_flag_reg;
// 中断请求
assign interrupt = timer_f lag_reg;
endmodule
```

18.6 文献备注

本章的文献备注信息和第 15 章节到第 17 章类似。

18.7 实验

18.7.1 可选择的计时器中断服务程序

示例 18.1 的中断服务程序用了两个专门的寄存器来记录时钟周期的个数。这两个寄存器不能被用来做其他的事情。可选择的方案是用一个 2bit 的数据

RAM 来实现这个目的，同时在这个服务程序里面使用临时性的寄存器。考虑到一个中断可以发生在任何时刻，因此我们必须保存和复原相应的寄存器。举一个例子来说，当这个计算完成的时候，如果在这个服务程序里面 S0 和 S1 寄存器被用来计算，它们的内容必须被保存，当服务程序被调用，然后被复原。得到汇编的 HDL 代码，编译和综合这个回路，验证其可操作性。

18.7.2　可编程的计时器

我们可以替代 18.5 节中的 mod_500 计数器，用一个通用的 mod_m 计数器，然后就可以让计时器实现可编程。这个新的计时器操作如下：

- m 是一个 12 位的无符数据；
- m 的 4 个 LSB 是 1111；
- 这个计时器有一个 8 位的寄存器来保存 m 的 MSB 高 8 位。寄存器被处理过，作为 PicoBlaze 的一个新的输出端口；
- 一个新的加载寄存器的控制按钮。当按下的时候，PicoBlaze 输入这个重要信息，源自于 8 位开关，输出这个重要信息到计时器的寄存器里面。

设计这个新的 I/O 接口，得到汇编的 HDL 代码，编译，综合这个回路。在计时器加载不同的重要信息，分析 LED 显示器发生了什么。

18.7.3　设置按钮中断服务程序

正如 17.4 节所讨论的方形回路，这个按钮被用来加载 8 位开关 a 和 b 的操作。它的状态是连续的，在主回路里面。我们可以修订部分代码，用一个中断机制来完成这个任务。这个中断服务程序包括几个临时的寄存器，然后它们必须在适当的时候被保存或复原，这在实验 18.7.1 里面已经讨论过。设计这个新的 I/O 接口，得到汇编 HDL 代码，编译，综合这个回路，验证其可操作性。

18.7.4　两个请求的中断服务程序

假定我们可以执行示例 18.1 中的每一个计时器的中断请求和实验 18.7.3 里 PicoBlaze 系统里的设置按钮中断请求。接着讨论 18.3.2 节，来设计新的中断接口和中断服务程序。得到一个汇编的 HDL 代码，编译，综合这个回路，验证其可操作性。

18.7.5　4 个请求的中断控制器

一个中断控制器帮助处理机来处理多重中断请求。4 个请求中断控制器的阻塞图，如图 18-6 所示。中断控制器应该包含 4 个标记 FF，一个特殊的优先编码回路。如果一个或多个中断请求是激活的，这个控制器就可以决定哪一个请求有

最高的优先权执行，代替它的2位代码在 req_id 端口，产生 int 信号。当 PicoBlaze 产生 interrupt_ack 信号的时候，这个控制器可以清除掉相应的标记 FF。简单来说，我们可以假定 int_request_3 有最高的优先权，int_request_0 有最低的优先权。

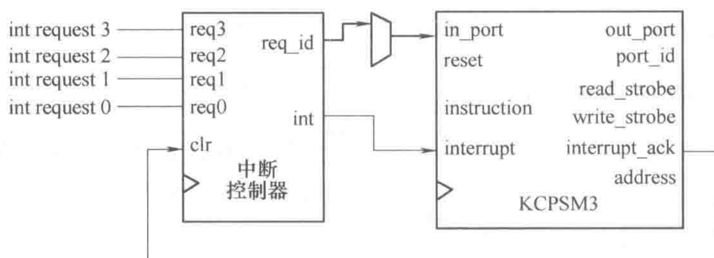


图 18-6　中断接口，用4个请求的中断处理

得到中断控制器的 HDL 代码，用这个新的控制器，重复 18.7.4 节的实验（这两个未使用的正确请求可以被约束为0）。

附录　Verilog 举例

A.1　数值和运算符

A.1.1　有符号数和无符号数

数	数　　值	注　　释
5'b11010	11010	
5'b11 _010	11010	_ 间隔符
5'032	11010	
5 'h1a	11010	
5'd26	11010	
5'b0	00000	0 扩展
5'b1	00001	0 扩展
5'bz	ZZZZZ	z 扩展
5'bx	XXXXX	x 扩展
5'bx01	xxx01	x 扩展
-5'b00001	11111	00001 的补码
'b11010	00000000000000000000000000011010	扩展到 32 位
'hee	00000000000000000000000011101110	扩展到 32 位
1	00000000000000000000000000000001	扩展到 32 位
-1	11111111111111111111111111111111	扩展到 32 位

A.1.2　运算符

运算符类型	符号	简单说明	操作数
算术运算符	+	加法	2
	-	减法	2
	*	乘法	2
	/	除法	2
	%	取模	2
	* *	求冥	2

（续）

<table>
<tr><th>运算符类型</th><th>符号</th><th>简单说明</th><th>操作数</th></tr>
<tr><td rowspan="4">移位运算符</td><td>> ></td><td>逻辑右移</td><td>2</td></tr>
<tr><td>< <</td><td>逻辑左移</td><td>2</td></tr>
<tr><td>> > ></td><td>算术右移</td><td>2</td></tr>
<tr><td>< < <</td><td>算术左移</td><td>2</td></tr>
<tr><td rowspan="4">关系运算符</td><td>></td><td>大于</td><td>2</td></tr>
<tr><td><</td><td>小于</td><td>2</td></tr>
<tr><td>> =</td><td>大于等于</td><td>2</td></tr>
<tr><td>< =</td><td>小于等于</td><td>2</td></tr>
<tr><td rowspan="4">相等运算符</td><td>= =</td><td>逻辑相等</td><td>2</td></tr>
<tr><td>！ =</td><td>逻辑不等</td><td>2</td></tr>
<tr><td>= = =</td><td>全等</td><td>2</td></tr>
<tr><td>！ = =</td><td>全不等</td><td>2</td></tr>
<tr><td rowspan="4">按位运算符</td><td>~</td><td>按位非</td><td>1</td></tr>
<tr><td>&</td><td>按位与</td><td>2</td></tr>
<tr><td>|</td><td>按位或</td><td>2</td></tr>
<tr><td>^</td><td>按位异或</td><td>2</td></tr>
<tr><td rowspan="3">规约运算符</td><td>&</td><td>归约与</td><td>1</td></tr>
<tr><td>|</td><td>归约或</td><td>1</td></tr>
<tr><td>^</td><td>归约异或</td><td>1</td></tr>
<tr><td rowspan="3">逻辑运算符</td><td>!</td><td>逻辑非</td><td>1</td></tr>
<tr><td>&&</td><td>逻辑与</td><td>2</td></tr>
<tr><td>||</td><td>逻辑或</td><td>2</td></tr>
<tr><td rowspan="2">连接运算符</td><td>{ }</td><td>连接</td><td>任意</td></tr>
<tr><td>{ { } }</td><td>复制</td><td>任意</td></tr>
<tr><td>条件运算符</td><td>? :</td><td>条件</td><td>3</td></tr>
</table>

A. 2 一般的 Verilog 构造

A. 2. 1 全部代码的组成

示例 A. 1 全部代码的组成

```
module bin-counter
```

```
// 可更改参数声明
#(parameter N =8) // 默认值为8
// 端口声明
(
  input wire clk,reset,                    // 时钟和复位
  input wire syn_clr,load,en,              // 输入控制信号
  input wire[N_1:0]d,                      // 输入数据
  output wire max_tick,                    // 输出状态
  output wire[N-1:O] q                     // 输出数据
);
// 定义常量
localparam MAX =2 * *N - 1;
// 定义信号
reg[N-1:0] r_reg, r_next;
// 实体
// ==============================================
// 例化部分
// ==============================================
// 这组代码中没有例化
// ==============================================
// 存储原理
// ==============================================
// 寄存器
always @ ( p o s e d g e clk, posedge reset)
        if (reset)
           r_reg < =0;
        else
           r_reg < =r_next;
// ==============================================
// 组合电路
// ==============================================
// 下一个状态逻辑
always @ *
        if (syn_clr)
           r_next =0;
```

```
        else if (load)
          r_next = d;
        else if (en)
          r_next = r_reg + 1;
        else
          r_next = r_reg;
        // 输出逻辑
        assign q = r_reg;
        assign max_tick = (r_reg = =2 * * N-1) ? 1'bl : 1'b0
    endmodule
```

A. 2. 2 例化部分

示例 A. 2 例化部分模版

```
module counter-inst
    (
    input wire clk, reset,
    input wire syn_clr16, load16, en16,
    input wire[15:0] d,
    output wire max_tick8, max_tick16,
    output wire[15:0] q
    );
    // 实体
    // 例化16 位计数器,所有用到的端口
    Bin_counter #(. N(16)) counter_16_unit
    (. clk(c1k),. reset(reset),
    . syn_clr(syn_clr16),. load(load16),. en(en16),
    . d(d),. max_tick(max_tick16),. q(q));
    // 例化8 位无限计数器
    // 仅用到max_tick 信号
    bin-counter counter-8-unit
    (. clk(clk),. reset(reset),
    . synclr(1'b0),. load(1'b0),. en(1'b1),
    . d(8'h00),. max_tick(max_tick8),. q());
```

```
endmodule
```

A.3　条件运算符操作以及 if 和 case 语句

A.3.1　条件运算符操作和 if 语句

示例 A.3　用条件运算符和 if 语句设计的优先译码器

```
    (
    input wire[4 : 1] r,
    output wire[2:0] y1,
    output reg[2:0] y2
    );
    // 条件控制符操作
    assign y1 = (r[4]) ? 3'b100 :// 也能用(r[4] = =1 'b1 )
                (r[3]) ? 3'b011 :
                (r[2]) ? 3'b010 :
    //If 语句
    // 每条分支能够包含多条语句
    // 用begin …end 分段
    always @ *
        if (r[4])
            y2 =3'b100;
        else if (r[3])
            y2 =3'b011;
        else if (r[2])
            y2 =3'b010;
        else if (r[1])
            y2 =3'b001;
        else
            y2 =3'b000;
endmodule
```

A. 3. 2　case 语句

示例 A. 4　用 case 语句设计的优先译码器

```
module prio_encoder_case
    (
    input wire[4:1]r ,
    output reg[2:0]y1,y2
    );
    //case 语句
    // 每条分支能够包含多条语句
    // 用begin …end 分段
    always @ *
        case (r)
            4'b1000, 4'b1001, 4'b1010, 4'b1011,
            4'b1100, 4'b1101, 4'b1110, 4'b1111:
                yl =3'b100;
            4'b0100, 4'b0101, 4'b0110, 4'b0111:
                yl =3'b011;
            4'b0010, 4'b0011:
                yl =3'b010;
            4'b0001:
                yl =3'b001;
            4'boo00 : // 也可以用default
                yl =3'b000;
        endcase
    //casez 语句
    always @ *
        casez (r)
            4'b1???: y2 =3'b100; // 表示无论为什么值
            4'b01??: y2 =3'b011;
            4'b001?: y2 =3'b010;
            4'b001: y2 =3'b001;
            4'b0000: y2 =3'b000;// 也可以用default
    endcase
```

```
endmodule
```

A.4 用 always 过程块组成的电路

A.4.1 过程块无默认输出值

示例 A.5 always 过程块模版(无默认输出值)

```
module compare_no_defult
    (
    input wire a,b,
    output reg gt, eq
    );
    // 用@ * 表示包含所有的输入敏感示例
    //else 分支不能被省略
    // 所有的输出在所有分支中必须赋值
    always @ *
        if (a > b)
          begin
            gt = 1'b1;
            eq = 1'b0;
          end
        else if(a = = b)
          begin
            gt = 1'b0;
            eq = 1'b1;
        end
        else//else 分支不能被省略
          begin
            gt = 1'b0;
            eq = 1'b0;
          end
    endmodule
```

A.4.2　过程块输出有默认值

示例 A.6　always 过程块模版(输出有默认值)

```
module compare_with_default
    (
    input wire a,b,
    output reg gt, eq
    );
    // 用@ * 表示包含所有的输入敏感示例
    // 给每个输出赋初值
    always @ *
        begin
          gt = 1'b0;// 给gt 赋初值
          eq = 1'b0;// 给eq 赋初值
          if (a > b)
            gt = 1'b1;
          else if(a == b)
            eq = 1'b1;
        end
    endmodule
```

A.5　存储组成

A.5.1　寄存器模版

示例 A.7　寄存器模版

```
module reg_template
    (
    input wire clk,reset,
    input wire en,
    input wire[7:0] q1_next,q2_next,q3_next,
    output reg[7:0] q1_reg,q2_reg,q3_reg
```

```
);
// ==========================================
// 寄存器未复位
// ==========================================
// 用非阻塞赋值
always @ (posedge clk)
     ql_reg < =ql_next;
// ==========================================
//// 异步复位的寄存器
// ==========================================
// 用非阻塞赋值
always @ (posedge clk,posedge reset)
     if (reset)
          q2_reg < =8'b0;
     else
          q2_reg < =q2_next;
// ==========================================
// 有使能端和异步复位的寄存器
// ==========================================
// 用非阻塞赋值
always @ (posedge clk,posedge reset)
     if (reset)
          q3_reg < =8'b0;
     else if (en)
          q3_reg < =q3_next;
endmodule
```

A.5.2 寄存器文件

示例 A.8 寄存器文件

```
module reg_file
   #(
   parameter B =8,// 数据位数
           W =2// 地址位数
```

```
    )
    (
    input wire clk,
    input wire wr_en,
    input wire[W-1:0] w_addr,r_addr,
    input wire[B-1:0] w_data,
    output wire[B-1:0]r_data
    );
    //信号定义
    reg[B-1:0]array_reg[2**W-1:0];
    //实体
    //写操作
    always @(posedge clk)
        if (wr_en)
            array_reg[w_addr] <=w_data;
    //读操作
    assign r_data=array_reg[r_addr];
endmodule
```

A.6　时序电路

示例 A.9　时序电路模版

```
//-----------------------------------------------
//Universal counter function table
//-----------------------------------------------
//syn_clr   load   en    q*        operation
//-----------------------------------------------
//     1      -     -     0        synchronous clear
//     0      1     -     d        parallel load
//     0      0     1     q+1      count up
//     0      0     0     q        pause
//-----------------------------------------------
module bin_counter
```

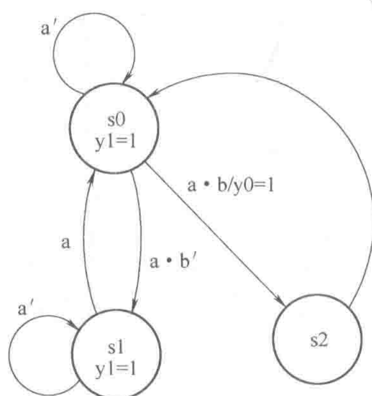
```
#(parameter N =8)// 默认值为8
(
input wire clk, reset,                // 时钟和复位
input wire syn_clr, load, en,         // 输入控制
input wire[N-1:0]d,                   // 输入数据
output wire max_tick,                 // 输出状态
output wire[N-1:0]q                   // 输出数据
);
// 数据定义
localparam MAX =2 * * N-1;
// 信号定义
reg[N-1:0]r_reg, r_next;
// 实体
// ========================================
// 寄存器
// ========================================
// 寄存器
always @ (posedge clk, posedge reset )
    if (reset)
        r_reg  < =0;
    else
        r_reg  < =r_next;
// ========================================
// 下一状态逻辑
// ========================================
always @ *
    if (syn_clr)
        r_next =0;
    else if(load)
        r_next =d;
    else if (en)
        r_next t =r_reg +1;
    else
        r_next =r_reg;
// ========================================
```

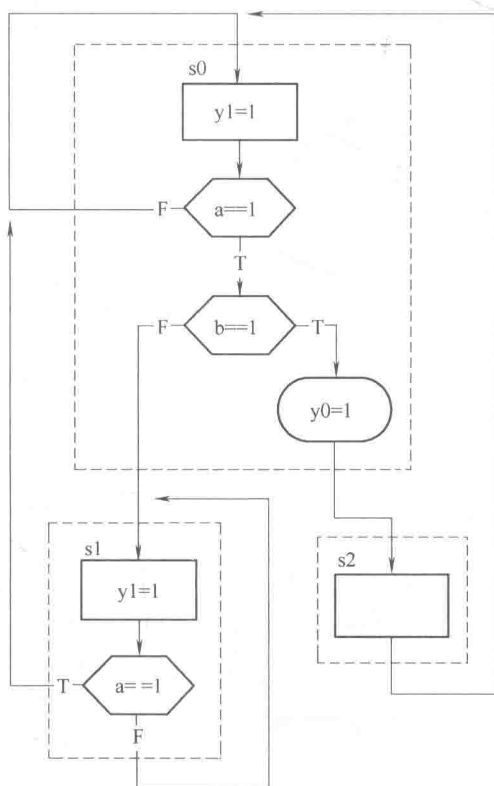
```
   // 输出逻辑
   // =======================================
   assign q = r_reg;
   assign max_tick = (r_reg = =2 * * N-1) ? 1'b1 : 1'b0;
endmodule
```

A.7 有限状态机

示例 A.10 有限状态机模版

```
   // 图 A.1 中的有限状态机代码
   module fsm_eg_2_seg
      (
```





a) 状态转换图 b) ASM流程图

图 A-1 一个有限状态机模版的状态转换图和 ASM 流程图

```
    input wire clk, reset,
    input wire a, b,
    output reg yo, yl
);
// 状态符号定义
localparam[1:0]s0 =2Ib00,
               s1 =2'b01,
               s2 =2'b10;
// 信号定义
reg[1: 0] state_reg, state_next;
// 状态寄存器
always @ (posedge clk,posedge reset)
    if(reset)
      state_reg < =s0;
    else
      state_reg < =state_next;
// 下一个状态逻辑和输出逻辑
always @ *
begin
    state_next = state_reg;
                        // 默认下一个状态不变
y1 =1'b0;               // 默认输出为0
y0 =1'b0;               // 默认输出为0
    case (state_reg)
      s0: begin
          y 1 =1'b1;
          if (a)
          if (b)
            begin
              state_next = s2;
              yo =1'b1;
            end
          else
            state_next = sl;
      end
```

```
        s1 : begin
            y1 = 1'b1;
            if(a)
                state_next = s0;
          end
        s2 : state_next = s0;
        default : state_next = s0;
      endcase
    end
endmodule
```

A.8 有限状态机数据

示例 A.11　FSMD 模板

```
// 在图A.2 中的FSMD 的代码
module fib
    (
    input wire clk, reset,
    input wire start,
    input wire[4:0] i,
    output reg ready,done_tick,
    output wire[19:0] f
    );
    // 状态符号定义
    localparam[1:0]
    idle = 2'b00,
    op = 2'b01,
    done = 2'b10;
    // 信号定义
    reg[1:0] state_reg, state_next;
    reg[19:0] t0_reg, t0_next, t1_reg, t1_next;
    reg[4:0] n_reg, n_next;
    // 实体
```

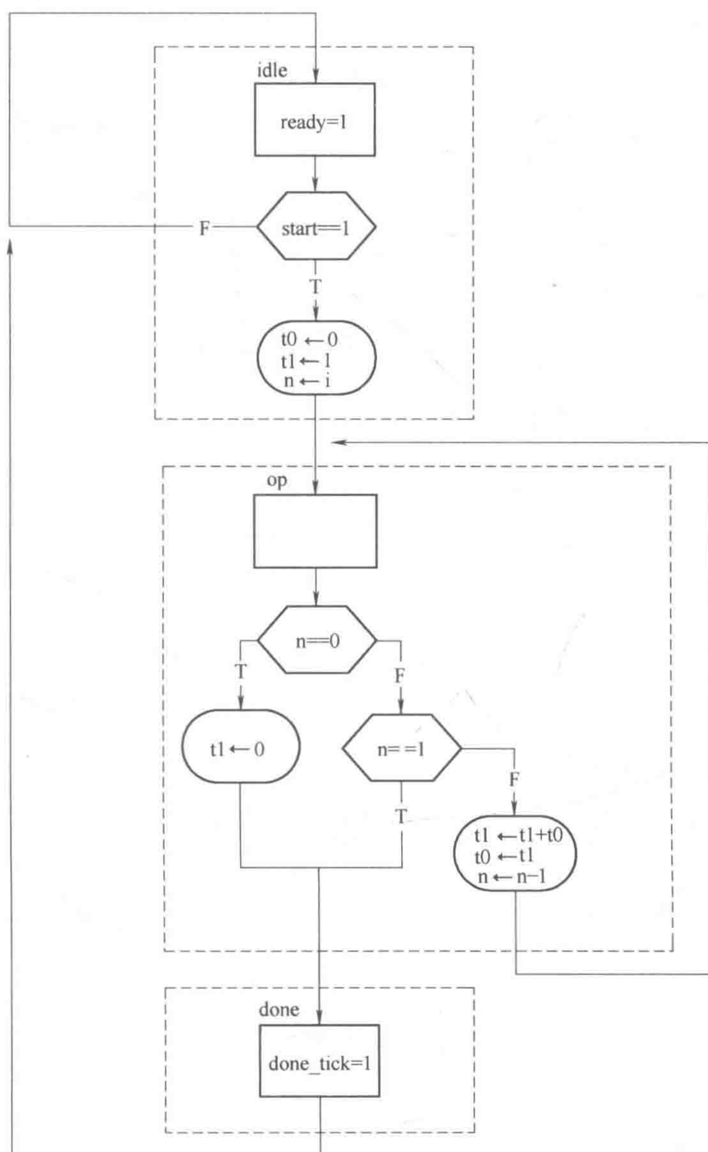


图 A-2 一个 FSMD 模版的 ASMD 流程图

```
// 状态和数据寄存器
always @ ( posedge clk , posedge reset)
    if( reset)
      begin
        state_reg  < = idle;
```

```
         t0_reg  < =0;
         t1_reg  < =0;
         n_reg  < =0;
      end
   else
      begin
         state_reg  < =state_next;
         t0_reg  < =t0_next;
         t1_reg  < =t1_next;
         n_reg  < =n_next;
      end
// 下一个状态逻辑和数据传输功能单元
always @ *
   begin
      state_next = state_reg;// 默认状态不变
      ready = 1'b0;          // 默认输出为0
      done_tick = 1'b0;      // 默认输出为0
      t0_next = t0_reg;      // 保持上一个值
      t1_next = t1_reg;      // 保持上一个值
      n-next = n_reg;        // 保持上一个值
      case (state_reg)
        idle:
          begin
              ready = l'dl;
              if (start)
              begin
                 t0_next = 0;
                 t1_next = 20'dl;
                 n-next = i;
                 state_next = op;
              end
          end
      op :
          if (n_reg = =0)
            begin
```

```
                t1_next = 0;
                state_next = done;
             end
          else if (n_reg = = 1)
                state_next = done;
          else
             begin
                t1_next = t1_reg + t0_reg;
                t0_next = t1_reg;
                n_next = n_reg - 1;
             end
      done :
          begin
             done_tick = 1'b1;
             state_next = id1e;
          end
        default : state_next = idle;
      endcase
   end
 // 输出
 assign f = t1_reg;
endmodule
```

A.9 S3 开发板的约束文件 S3. UCF

```
# ==============================================
#引脚分配
#Spartan-3 Starter 开发板
# ==============================================
#时钟和复位
# ==============================================
NET "clk"      LOC = "T9" ;
NET "reset"    LOC = "L14" ;
#按钮和开关
```

```
# ================================================================
#4 个按钮
NET "btn<0>"      LOC = "M13";
NET "btn<1>"      LOC = "M14";
NET "btn<2>"      LOC = "L13";
# NET "btn<3>"    LOC = "L14"; #btn<3>也可以用作复位
#8 个滑动开关
NET "sw<0>"       LOC = "F12";
NET "sw<1>"       LOC = "G12";
NET "sw<2>"       LOC = "H14";
NET "sw<3>"       LOC = "H13";
NET "sw<4>"       LOC = "J14";
NET "sw<5>"       LOC = "J13";
NET "sw<6>"       LOC = "K14";
NET "sw<7>"       LOC = "K13";
# ================================================================
# RS232
# ================================================================
NET "rx"          LOC = "T13" | DRIVE = 8 | SLEW = SLOW;
NET "tx"          LOC = "R13" | DRIVE = 8 | SLEW = SLOW;
# ================================================================
#4 个数字多路七段发光二
#极管显示器
# ================================================================
#数字使能
NET "an<0>"       LOC = "D14";
NET "an<1> "      LOC = "G14";
NET "an<2> "      LOC = "F14";
NET "an<3> "      LOC = "E13";
#7 段发光二极管分配
NET "sseg<7>"     LOC = "P16";#decimal  point
NET "sseg<6>"     LOC = "E14";#segment  a
NET "sseg<5>"     LOC = "G13";#segment  b
NET "sseg<4>"     LOC = "N15";#segment  c
NET "sseg<3>"     LOC = "P15";#segment  d
```

```
NET "sseg<2>"    LOC = "R16" ;#segment  e
NET "sseg<1>"    LOC = "F13" ;#segment  f
NET "sseg<0>"    LOC = "N16" ;#segment  g
# ==================================================
#8 个不连续的发光二极管
# ==================================================
NET "led<0>"     LOC = "K12" ;
NET "led<1>"     LOC = "P14" ;
NET "led<2>"     LOC = "L12" ;
NET "led<3>"     LOC = "N14" ;
NET "led<4>"     LOC = "P13" ;
NET "led<5>"     LOC = "N12" ;
NET "led<6>"     LOC = "P12" ;
NET "led<7>"     LOC = "P11" ;
# ==================================================
# 视频图形阵列
# ==================================================
NET "rgb<2>"     LOC = "R12" | DRIVE = 8 | SLEW = FAST;
NET "rgb<1>"     LOC = "T12" | DRIVE = 8 | SLEW = FAST;
NET "rgb<0>"     LOC = "R11" | DRIVE = 8 | SLEW = FAST;
NET "vsync "     LOC = "T10" | DRIVE = 8 | SLEW = FAST;
NET "hsync "     LOC = "R9"  | DRIVE = 8 | SLEW = FAST;
# ==================================================
#PS2 端口
# ==================================================
NET "ps2c"       LOC = " M16" | IOSTANDARD = LVCMOS33 | DRIVE = 8
ISLEW = SLOW;
NET "ps2d1"      LOC = " M15" | IOSTANDARD = LVCMOS33 | DRIVE = 8
ISLEW = SLOW;
// ==================================================
# 两片静态存储器
// ==================================================
# 共用的18 位存储地址
NET "ad<17>"     LOC = "L3" | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<16>"     LOC = "KS" | IOSTANDARD = LVCMOS33 | SLEW = FAST;
```

```
NET "ad<15>"      LOC = "K3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<14>"      LOC = "J3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<13>"      LOC = "J4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<12>"      LOC = "H4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<11>"      LOC = "H3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<10>"      LOC = "GS"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<9>"       LOC = "E4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<8>"       LOC = "E3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<7>"       LOC = "F4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<6>"       LOC = "F3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<5>"       LOC = "G4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<4>"       LOC = "L4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<3>"       LOC = "M3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<2>"       LOC = "M4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<1>"       LOC = "N3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ad<0>"       LOC = "L5"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
#共用的读写使能
NET "oe_n"        LOC = "K4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "we_n"        LOC = "G3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
#静态存储芯片1 的数据总线,片选,高位,低位
NET "dio_a<15>"   LOC = "R1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<14>"   LOC = "P1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<13>"   LOC = "L2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<12>"   LOC = "J2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<11>"   LOC = "H1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<10>"   LOC = "F2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<9>"    LOC = "P8"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<8>"    LOC = "D3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<7>"    LOC = "B1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<6>"    LOC = "C1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<5>"    LOC = "C2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<4>"    LOC = "RS"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<3>"    LOC = "T5"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<2>"    LOC = "R6"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_a<1>"    LOC = "T8"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
```

```
NET "dio_a〈0〉"    LOC = "N7"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ce_a_n"       LOC = "P7"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ub_a_n"       LOC = "T4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "lb_a_n"       LOC = "P6"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
#静态存储芯片1 的数据总线,片选,高位,低位
NET "dio_b〈15〉"   LOC = "N1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈14〉"   LOC = "M1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈13〉"   LOC = "K2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈12〉"   LOC = "C3"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈11〉"   LOC = "F5"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈10〉"   LOC = "GI"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈9〉"    LOC = "E2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈8〉"    LOC = "D2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈7〉"    LOC = "D1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈6〉"    LOC = "E1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈5〉"    LOC = "G2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈4〉"    LOC = "J1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈3〉"    LOC = "K1"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈2〉"    LOC = "M2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "dio_b〈1〉"    LOC = "N2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "die_b〈0〉"    LOC = "P2"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ce_b_n"       LOC = "N5"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "ub_b_n"       LOC = "R4"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
NET "lb_b_n"       LOC = "P5"  | IOSTANDARD = LVCMOS33 | SLEW = FAST;
# ==================================================
#时序约束S3 板上的50MHz
#振荡器
#时钟信号的名称是clk
# ==================================================
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 40 ns HIGH 50 %;
```

参考文献

[1] P. J. Ashenden, The Designer's Guide to VHDL, 2nd ed. , Morgan Kaufmann, 2001.

[2] J. Axelson, Serial Port Complete, 2nd ed. , Lakeview Research, 2007.

[3] L. Bening and H. D. Foster, Principles of Verifiable RTL Design, 2nd ed. , Springer-Verlag, 2001.

[4] J. Bergeron, Writing Testbenches: Functional Verifiable of HDL Models, Springer-Verlag, 2003.

[5] K. Chapman, "Creating Embedded Microcontrollers," TechXclusives at http://www. Xilinx. com.

[6] A. Chapweske, "PS/2 Mouse/Keyboard Protocol," http://www. computer-engineering. org.

[7] A. Chapweske, "PS/2 Keyboard Interface," http://www. computer-engineering. org.

[8] A. Chapweske. "PS/2 Mouse Interface," http://www. computer-engineering. org.

[9] P. P. Chu, RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability, Wiley-IEEE Press, 2006.

[10] M. D. Ciletti, Advanced Digital Design with the Verilog HDL, Prentice Hall, 2003.

[11] M. D. Ciletti, Starter's Guide to Verilog 2001, Prentice Hall, 2003.

[12] C. E. Cummings, ""full_case parallel-case", the Evil Twins of Verilog Synthesis," SNUG (Synopsys Users Group Conference), Boston, 1999.

[13] C. E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!" SNUG (Svnops-vs Users Group Conference), San Jose, 2000.

[14] C. E. Cummings, "Coding and Scripting Techniques for FSM Designs with Synthesis-Optimized, Glitch-Free Outputs," SNUG(Synopsys Users Group Conference), Boston, 2000.

[15] C. E. Cummings, "New Verilog-2001 Techniques for Creating Parameterized Models (or Down With' define and Death of a defparam!)," International HDL Conference, 2002.

[16] D. D. Gajski, Principles of Digital Design, Prentice Hall, 1997.

[17] J. O. Hamblenetal. , Rapid Prototyping of Digital Svstems: Quartus8 II Edition, Springer, 2005.

[18] IEEE, IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364-2001), Institute of Electrical and Electronics Engineers, 2001.

[19] IEEE, IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-2001), Institute of Electrical and Electronics Engineers, 2001.

[20] M. Keating and P. Bricaud, Methodology Manual for System-on-a-Chip Designs, 3rd ed. , Springer-Verlag, 2002.

[21] C. M. Maxfield, The Design Warrior's Guide to FPGAs, Newnes, 2004.

[22] Mentor Graphics, ModelSim Tutorial, Mentor Graphics Corporation.

[23] S. Palnitkar, Verilog HDL, 2nd ed. , Prentice Hall, 2003.

[24] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd ed. , Morgan Kaufmann, 2004.

[25] J. M. Rabaey, Digital Integrated Circuits, 2nd ed. , Prentice Hall, 2002.

[26] S. Sutherland, "What's New in the IEEE 1364 Verilog-2001 Standard," International HDL Conference,2000.

[27] J. F. Wakerly, Digital Design: Principles and Practices, Prentice Hall, 2002.

[28] W. Wolf, FPGA-Based System Design, Prentice Hall, 2004.

[29] Xilinx, DS099 Spartan-3 FPGA Family: Complete Data Sheet, Xilinx, Inc.

[30] Xilinx, ISE 8. li Quick Start Tutorial, Xilinx, Inc.

[31] Xilinx, ISE In-Depth Tutorial, Xilinx, Inc.

[32] Xilinx, PicoBlaze 8-Bit Embedded Microcontroller User Guide, Xilinx, Inc.

[33] Xilinx, Spartan-3 Starter Kit Board User Guide, Xilinx, Inc.

[34] Xilinx, XAPP462 Using Digital Clock Managers(DCMs) in Spartan-3 FPGAs, Xilinx, Inc.

[35] Xilinx, XAPP463 Using Block RAM in Spartan-3 Generation FPGAs, Xilinx, Inc.

[36] Xilinx, XAPP464 Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs, Xilinx, Inc.

[37] Xilinx. XST User Guide v8. li, Xilinx, Inc.

国际视野 科技前沿

国际信息工程先进技术译丛

《用Verilog设计FPGA样机实例解析
《基于FSM和Verilog HDL的数字电
《移动协议与切换优化：设计、评估与
《全IP网络融合》
《不确定性理论与多传感器数据融合》
《无线通信系统中的定位技术与应用》
《基于大数据的商务智能分析》
《3GPP网络中的IPv6部署：从2G向LTE及未来移动宽带的演进》
《MIMO无线网络手册》
《可重构无线电系统的网络架构和标准》
《声学显微镜与超分辨率成像理论及应用》
《构建基于IPv6和移动IPv6的物联网：向M2M通信的演进》
《虚拟网络——下一代互联网的多元化方法》
《下一代融合网络理论与实践》
《认知视角下的无线传感器网络》
《移动云计算：无线、移动及社交网络中分布式资源的开发利用》
《Android系统安全与攻防》
《内容分发网络》
《计算机网络仿真OPNET实用指南》
《移动无线信道》（原书第2版）
《LTE-Advanced：面向IMT-Advanced的3GPP解决方案》
《声学成像技术及工程应用》
《LTE/SAE网络部署实用指南》
《认知无线电通信与组网：原理与应用》
《网络性能分析原理与应用》
《云连接与嵌入式传感系统》
《IP地址管理原理与实践》
《自组织网络：GSM,UMTS和LTE的自规划、自优化和自愈合》
《实现吉比特传输的60GHz无线通信技术》
《LTE自组织网络（SON）：高效的网络管理自动化》
《UMTS中的LTE：向LTE-Advanced演进》（原书第2版）
《UMTS中的WCDMA - HSPA演进及LTE（原书第5版）
《无线传感器及执行器网络》
《认知无线电网络》
《网络融合——服务、应用、传输和运营支撑》
《UMTS中的LTE:基于OFDMA和SC-FDMA的无线接入》
《吉规模集成电路互连工艺及设计》
《高性能微处理器电路设计》

WILEY





机械工业出版社微信公众号



机械工业出版社E视界



ISBN 978-7-111-53644-4
9 787111 536444

上架指导 工业技术 / FPGA设计

ISBN 978-7-111-53644-4

定价：165.00元

[General Information]
书名=用Verilog设计FPGA样机实例解析　XilinxSpartan　第3版
作者=（美）彭皮·楚（Pong P.Chu）著
页数=553
SS号=14154275
DX号=
出版日期=2016.11
出版社=北京机械工业出版社